ADJ-702-137C

日立マイクロコンピュータ開発環境システム
H8S，H8/300 シリーズ
C コンパイラ　ユーザーズマニュアル

HSS008CLCS2S

H8S，H8/300 シリーズ　C コンパイラ　ユーザーズマニュアル
発行年月日　平成 4 年 9 月　第 1 版
平成 7 年 9 月　第 3 版
平成 9 年 8 月　第 4 版
発行　株式会社　日立製作所
半導体事業部
編集　株式会社日立マイコンシステム
技術情報センタ
©株式会社　日立製作所　1992

1. 本資料に記載された製品および製品の仕様は、信頼性、機能、設計の改良の理由により予告なく変更されることがあります。
2. 本資料の一部または全部を当社に無断で転載または複製することを堅くお断りします。
3. 本資料に記載された情報は、正確かつ信頼し得るものであります。ただし、これら記載された情報、製品または回路の使用に起因する損害または特許権その他権利の侵害に関しては、(株)日立製作所は一切その責任を負いません。
4. 本資料によって第三者または(株)日立製作所の特許権その他権利の実施権を許諾するものではありません。

5 日立製半導体は、特に医療機器用として推奨できる製品を用意しておりません。しかしながら、医療機器用として日立製半導体をお考えのお客様は、当社営業窓口へ是非ご相談頂きますようお願いいたします。

はじめに

本マニュアルは、「H8S, H8/300 シリーズ C コンパイラ」の使用方法を述べたものです。

本 C コンパイラは、C 言語で記述したソースプログラムを日立シングルチップマイクロコンピュータ H8S/2600 シリーズ、H8S/2000 シリーズ、H8/300H シリーズ、H8/300 シリーズ、または H8/300L シリーズのオブジェクトプログラムに変換するソフトウェアシステムです。H8S, H8/300 シリーズ C コンパイラをご使用になる前に、本マニュアルを良く読んで理解してください。

本マニュアルは本文 6 章と付録で構成されています。各章の内容を以下に示します。

第 1 章　概要・操作

まず概要として、C コンパイラの機能について説明します。

つづいて操作方法として、［1］C コンパイラ実行の手順、［2］オプション機能、［3］コンパイルリストの見方について説明します。

第 2 章　プログラミング

C コンパイラの限界値、オブジェクトプログラムの実行方式などプログラム開発時に考慮すべき事項について説明します。

第 3 章　システム組み込み

C コンパイラの生成したオブジェクトプログラムをシステムに組み込むために必要な事項として、メモリ割り付け手法や ROM 化の方法について説明します。また、標準入出力ライブラリ、メモリ管理ライブラリを使用する場合にユーザが作成しなければならない低水準インタフェースルーチンの仕様について説明します。

第 4 章　C コンパイラのエラーメッセージ

コンパイル時に発生するエラーメッセージとエラー内容、実行時に発生する C ライブラリ関数のエラーメッセージとエラー内容を説明します。

第 5 章　モジュール間最適化

C コンパイラおよびアセンブラ出力オブジェクトプログラムを、リンク時にモジュール間にまたがって最適化するモジュール間最適化ツールの機能、使用方法について説明します。

第 6 章　モジュール間最適化のエラーメッセージ

モジュール間最適化ツール実行時に発生するエラーメッセージとエラー内容を説明します。

■注意

H8S, H8/300 シリーズ C コンパイラ（Ver. 1.0）から使用している場合は、**「付録 H　旧バージョンとの相違点」**を参照してください。

■表記上の注意事項

本マニュアルの説明の中で用いられる記号は、次の意味を示しています。

〈 〉	**この記号で囲まれた内容を指定することを示します。**
[ ]	**省略してもよい項目を示します。**
． ． ．	**直前の項目を 1 回以上指定することを示します。**
△	**1 個以上の空白を示します。**
（RET）	**キャリッジリターンキー（リターンキーともいいます）を示します。**
|	**| で区切られた項目を選択できることを示します。**
（CNTL）	**次の文字を、コントロールキーを押しながら入力することを示します。**

本マニュアルは UNIX[*1]、または PC-9801[*2]シリーズ、IBM PC[*3]およびその互換機上で動作する MS-DOS[*4]に対応するように書かれています。UNIX 上で動作するコンパイラを以下 UNIX 版と称します。MS-DOS 上で動作するコンパイラを以下 PC 版と称します。

【注】 *1： UNIX **は、**X/Open **カンパニーリミテッドがライセンスしている米国ならびに他の国における登録商標です。**

*2： PC-9801 **は、日本電気株式会社の商標です。**

*3： IBM PC **は、米国** International Business Machines Corporation **の登録商標です。**

*4： MS-DOS **は、米国マイクロソフト社の登録商標です。**

Ｃコンパイラバージョンアップにおける注意事項

Ｃコンパイラをバージョンアップしてプログラム開発される場合、プログラムの動作が変わることがあります。プログラムを作成する際は、以下の点に注意して、お客様のプログラムを十分にテストしてください。

(a) プログラムの実行時間やタイミングに依存するプログラム

Ｃ言語仕様は、プログラムの実行時間については何も規定していません。したがってコンパイラのバージョンの違いによりプログラムの実行時間と I/O 等周辺機器のタイミングのずれ、あるいは割り込み処理のような非同期処理の時間の差等により、プログラムの動作が変わる場合があります。

(b) 一つの式に 2 個以上の副作用が含まれているプログラム

一つの式に 2 個以上の副作用が含まれている場合、コンパイラのバージョンにより、動作が変わる可能性があります。

■例

```
a[i++]=b[i++];  /*i のインクリメント順序は不定です。*/
f(i++, i++);    /*インクリメントの順序でパラメタの値が変わります。*/
                /*i の値が 3 の時 f(3, 4) または f(4, 3) になります。*/
```

(c) 結果がオーバフローや不当演算に依存するプログラム

オーバフローが生じた場合や、不当演算を実施した場合、結果の値は保証しません。したがって、バージョンが変わると動作が変わる可能性があります。

■例

```
int a, b;
x=(a*b)/10; /*a と b の値の範囲によってはオーバフローする可能性があります*/
```

（d）変数の初期化抜け、型の不一致

変数が初期化されていない場合や、パラメタやリターン値の型が呼び出し側と呼び出される側で対応していない場合、不正な値をアクセスすることになります。したがって、コンパイラのバージョンによって動作が変わる場合があります。

■例

file 1:

```
int f(double d)
{
          :
}
```

:

file 2:

```
int g(void)
{
        f(1);
}
```

関数呼び出し側のパラメタはint型ですが、関数定義側のパラメタは、double型のため、値を正しく参照できません。

上記に記載された情報が全ての起こりうる状況を示したわけではありません。したがって、お客様の責任で本Ｃコンパイラを正しくご使用の上、お客様のプログラムを十分にテストしてください。

目次

第 1 章　概要･操作

1.1　概要 3
1.2　プログラムの開発手順 4
1.3　C コンパイラの実行 5
1.3.1　C コンパイラの起動方法 5
1.3.2　ファイル名の付け方 7
1.3.3　コンパイラオプション 8
1.3.4　コンパイラの環境変数 29
1.3.5　標準ライブラリとの対応 31
1.3.6　コンパイルリストの見方 32

第 2 章　プログラミング

2.1　C コンパイラの限界値 43
2.2　C プログラムの実行方式 45
2.2.1　オブジェクトプログラムの構造 45
2.2.2　データの内部表現 47
2.2.3　アセンブリプログラムとの結合 52
2.3　拡張機能 62
2.3.1　#pragma 拡張子 62
2.3.2　組み込み関数 81
2.3.3　簡易入出力関数のサポート 94
2.4　プログラム作成上の注意事項 95
2.4.1　コーディング上の注意事項 95
2.4.2　プログラム開発上の注意事項 97

第３章　システム組み込み

3.1. システム組み込みの概要 103
3.2. メモリ領域の割り付け 104
3.2.1 静的領域の割り付け 104
3.2.2 動的領域の割り付け 108
3.3 実行環境の設定 113
3.4 C ライブラリ関数の実行環境の設定 120

第４章　C コンパイラのエラーメッセージ

4.1 C コンパイラのエラーメッセージ 139
4.2 C ライブラリ関数のエラーメッセージ 160

第５章　モジュール間最適化

5.1 概要 165
5.2 プログラムの開発手順 167
5.3 最適化ツールの起動方法 168
5.4 オプション／サブコマンド 171
5.4.1 最適化内容指定 173
5.4.2 最適化抑止指定 176
5.4.3 最適化情報 178
5.4.4 サブコマンドファイル 179

第６章　モジュール間最適化のエラーメッセージ

6.1 モジュール間最適化のエラーメッセージ 183

付録

A. C コンパイラが規定する言語仕様と C ライブラリ関数仕様 195
A.1 言語仕様 195
A.2 C ライブラリ関数仕様 204
A.3 浮動小数点の仕様 209
B. 引数割り付けの具体例 218
B.1 H8S/2600 用、H8S/2000 用、H8/300H 用 218
B.2 H8/300 用 223
C. レジスタとスタック領域の使用法 226

C.1 H8S/2600 用、H8S/2000 用、H8/300H 用アドバンストモード 226
C.2 H8S/2600 用、H8S/2000 用、H8/300H 用ノーマルモード 228
C.3 H8/300 用 230
D. 終了処理関数の作成例 231
D.1 終了処理の登録と実行（onexit）ルーチンの作成例 231
D.2 プログラムの終了（exit）ルーチンの作成例 231
D.3 異常終了（abort）ルーチンの作成例 234
E. 低水準インタフェースルーチンの作成例 235
E.1 低水準インタフェースルーチン作成例 235
F. 短絶対アドレスのアクセス範囲 242
F.1 短絶対アドレスのアクセス範囲 242
G. CPU 情報ファイルの作成 243
G.1 CIA の機能 243
G.2 CIA の起動方法 243
G.3 CIA の使用手順と選択メニュー 244
G.4 CIA の使用例 246
G.5 制限事項一覧 247
G.6 CIA のエラーメッセージ 248
H. 旧バージョンとの相違点 249
H.1 機能追加・改善項目 249
H.2 言語仕様の拡張 251
I. ASCII コード一覧表 254
I.1 ASCII コード一覧表 254

1. 概要・操作

第 1 章　目次

1.1　概要 3
1.2　プログラムの開発手順 4
1.3　C コンパイラの実行 5
1.3.1　C コンパイラの起動方法 5
1.3.2　ファイル名の付け方 7
1.3.3　コンパイラオプション 8
1.3.4　コンパイラの環境変数 29
1.3.5　標準ライブラリとの対応 31
1.3.6　コンパイルリストの見方 32

1.1 概要

H8S, H8/300 シリーズ C コンパイラは、C 言語で記述したソースプログラムを入力し、H8S, H8/300 シリーズ用リロケータブルオブジェクトプログラムまたはアセンブリソースプログラムを出力します。

本 C コンパイラは、以下の CPU をサポートしています。

・H8S/2600 シリーズ（以下、H8S/2600 と略す）
・H8S/2000 シリーズ（以下、H8S/2000 と略す）
・H8/300H シリーズ（以下、H8/300H と略す）
・H8/300 シリーズ（以下、H8/300 と略す）
・H8/300L シリーズ（以下、H8/300 と略す）*1

また、H8S/2600、H8S/2000、H8/300H では、ノーマルモードとアドバンストモードの各 CPU／動作モードもサポートしています。

【注】 *1 H8/300L **シリーズは、**H8/300 **シリーズと同一の命令仕様なので、**C **コンパイラでは** H8/300 **シリーズと同一に扱います。**

1.2 プログラムの開発手順

本システムを用いたプログラムの開発手順を**図** 1.1 に示します。破線で囲まれた部分は、本システムが提供するソフトウェアを示します。

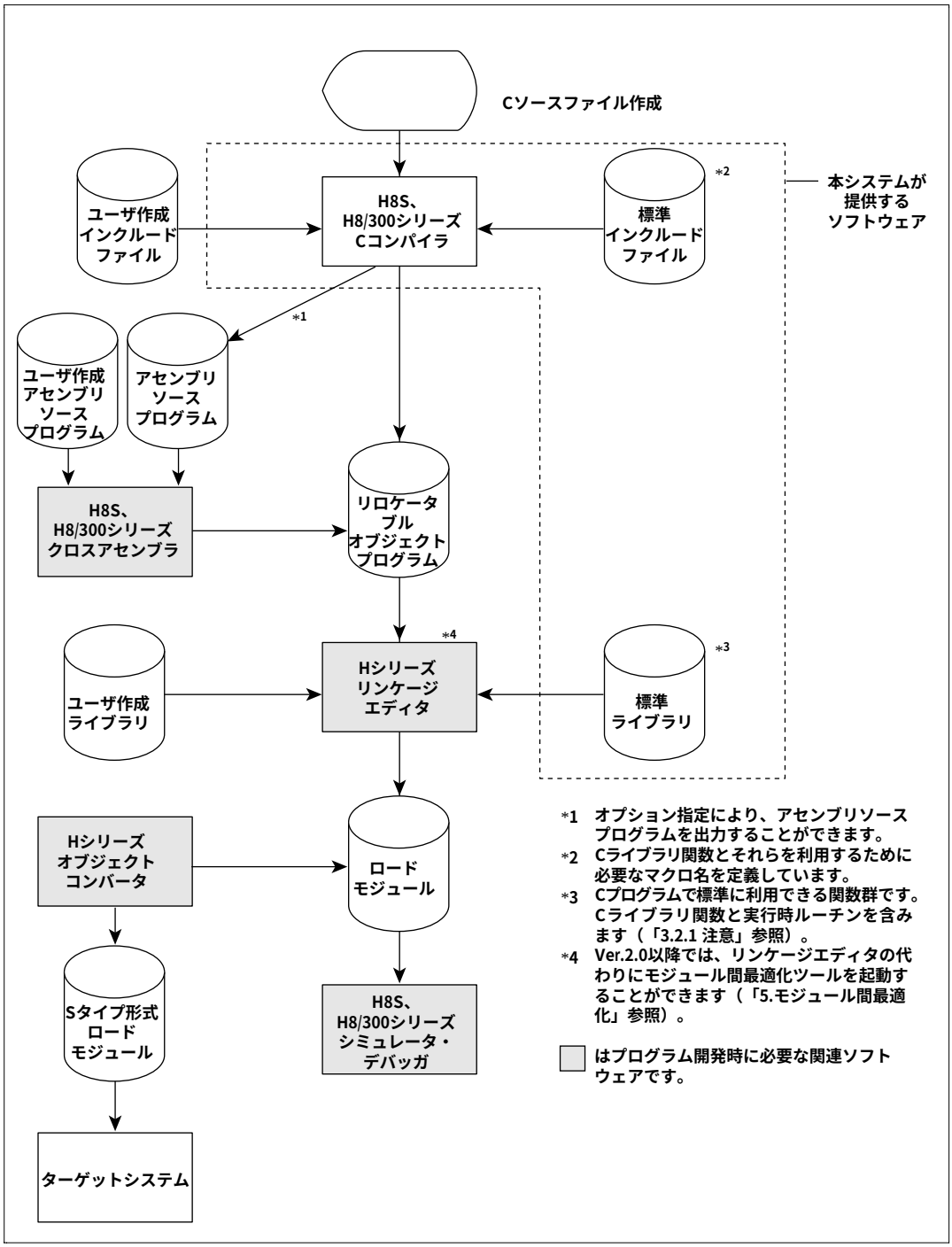


図 1.1 **プログラムの開発手順**

1.3 C コンパイラの実行

本節ではC コンパイラの起動方法、オプションの指定方法、コンパイルリストの見方について解説します。

1.3.1 C コンパイラの起動方法

C コンパイラを起動するコマンドラインの形式は次のとおりです。

ch38［△〈オプション〉．．．］［△〈ファイル名〉［△〈オプション〉．．．］．．．］

また、オプションの形式は次のとおりです。

〈オプション〉［＝〈サブオプション〉］［, 〈サブオプション〉］．．．］

以下、C コンパイラの基本的な操作方法を説明します。

(1) 起動コマンドの入力

```
ch38 (RET)
```

コンパイルは行なわず、コマンドライン形式、オプション一覧を出力します。

(2) プログラムのコンパイル

```
ch38△-cpu=2600a△test.c (RET)
```

C ソースプログラム「test.c」をコンパイルします。「-cpu=2600a」で cpu／動作モードを指定します。

C コンパイラは CPU／動作モードを必ず指定する必要があります。cpu オプションを省略した場合は、H38CPU 環境変数の内容を参照します。H38CPU 環境変数の指定方法については、**「1.3.4 コンパイラの環境変数」**を参照してください。

(3) オプション指定方法

```
ch38△-cpu=2600a△-debug△test.c (RET)
ch38△-cpu=2600a△-debug△-show=object,expansion△test.c (RET)
```

オプション cpu、debug、show の前に、-を付加します。

複数のオプションを指定するときは、スペース（△）で区切ります。

また、複数のサブオプションを指定するときはカンマ（, ）で区切ります。

■注意

MS-DOS 版では、オプションの前に-か/のいずれかを指定することができます。

また、複数のサブオプションを指定するときサブオプション列を（）でくくって指定することもできます。

（4）複数のプログラムのコンパイル

複数の C ソースプログラムを一度にコンパイルできます。

■例 1：複数プログラムの指定方法

```
ch38△-cpu=2600a△test1.c△test2.c (RET)
```

■例 2：オプションの指定（C ソースプログラムすべてに有効なオプション指定例）

```
ch38△-cpu=2600a△-object△test1.c △test2.c (RET)
```

test1.c、test2.c ともに object オプションが有効になります。

■例 3：オプションの指定（各 C ソースプログラムごとに有効なオプション指定例）

```
ch38△-cpu=2600a△test1.c△test2.c△-object (RET)
```

object オプションは test2.c のみ有効になります。C ソースプログラムごとのオプション指定は、C ソースプログラム全体に対するオプション指定よりも優先します。

1.3.2 ファイル名の付け方

ファイル名指定時に拡張子を省略した場合、C コンパイラは標準のファイル拡張子を付加したファイル名を使用します。C コンパイラおよび関連ソフトウェアで使用する標準のファイル拡張子を**表** 1.1 に示します。

表 1.1 C **コンパイラで使用する標準のファイル拡張子**

No.	**拡張子**	**意　　味**
1	c	**C 言語で記述されたソースプログラムファイル**
2	h	**インクルードファイル**
3	lis、lst	**リストファイル***1
4	obj	**リロケータブルオブジェクトプログラムファイル**
5	src	**アセンブリソースプログラムファイル**
6	lib	**ライブラリファイル**
7	abs	**アブソリュートロードモジュールファイル**
8	rel	**リロケータブルロードモジュールファイル**
9	map	**リンケージマップリストファイル**

【注】 *1 UNIX **版では** lis、MS-DOS **版では** lst **です。**

ファイル名の付け方の一般的な規則は、各ホストマシンに準じています。ご使用になるホストマシンのマニュアルを参照してください。

1.3.3 コンパイラオプション

コンパイラオプションの形式と短縮形および省略時解釈の一覧を**表** 1.2 に示します。下線部（__）は短縮形指定時の文字を示します。また、*斜体字*は省略時解釈を示します。

オプション、サブオプションの指定方法は**「**1.3.1 **C コンパイラの起動方法」**を参照してください。

表 1.2 **コンパイラオプション一覧**

No.	項目	形式	指定内容
1	CPU／**動作モード**	cpu = 2600n | 2600a [:<**アドレス空間のビット幅**>] | 2000n | 2000a [:<**アドレス空間のビット幅**>] | 300hn | 300ha[:<**アドレス空間のビット幅**>] | 300 | 300l | 300reg	H8S/2600 **ノーマルモードのオブジェクトを生成** H8S/2600 **アドバンストモードのオブジェクトを生成** H8S/2000 **ノーマルモードのオブジェクトを生成** H8S/2000 **アドバンストモードのオブジェクトを生成** H8/300H **ノーマルモードのオブジェクトを生成** H8/300H **アドバンストモードのオブジェクトを生成** H8/300 **のオブジェクトを生成**
2	**最適化レベル**	optimize = 0 | *1*	**最適化なしのオブジェクトを出力** **最適化ありのオブジェクトを出力**
3	**最適化内容**	speed [=register | shift | loop | switch | inline | struct]	**レジスタ退避・回復を** push,pop **で展開** **シフト演算をスピード優先でオブジェクト展開** **ループ文をスピード優先でオブジェクト展開** switch **文をスピード優先でオブジェクト展開** **サイズの小さい関数をインライン展開** **構造体代入式をスピード優先でオブジェクト展開**
4	**インライン展開**	nestinline = <**数値**> **省略時：***(nestinline=1)*	**インライン展開関数の、関数呼び出しネスト回数の指定**
5	**モジュール間最適化**	goptimize	**モジュール間最適化情報を出力**
6	**デバッグ情報**	debug *nodebug*	**出力あり** **出力なし**
7	**リスト内容と形式**	show = *source* | nosource | object | *noobject* | *statistics* | nostatistics | allocation | *noallocation* | expansion | *noexpansion* | width = <**数値**> | length = <**数値**> **省略時：**(*w = 0, l = 0*)	**ソースリストの有無** **オブジェクトリストの有無** **統計情報の有無** **シンボル割り付け情報リストの有無** **マクロ展開後リストの有無** **１行の最大文字数　　数値**: 0, 80〜132 **ページ内の最大行数　数値**: 0, 20〜255
8	**リストファイル**	*list* [= <**ファイル名**>] nolist	**出力あり** **出力なし**
9	**インフォメーションメッセージ出力**	message	**インフォメーションメッセージを出力**
10	**オブジェクトファイル**	*object* = <**ファイル名**> noobject	**出力あり** **出力なし**
11	**オブジェクト形式**	code = *machinecode* | asmcode	**機械語プログラムを出力** **アセンブリソースプログラムを出力**
12	**マクロ名の定義**	define = <**マクロ名**>=<**名前**> | <**マクロ名**>=<**定数**> | <**マクロ名**>	**<名前>を<マクロ名>として定義** **<定数>を<マクロ名>として定義** **<マクロ名>を定義したものと仮定**
13	**インクルードファイル**	include = <**パス名**>	**インクルードファイルの取り込み先パス名を指定（複数指定可）**

No.	項目	形式	指定内容
14	インクルードファイル基点ディレクトリ	chgincpath	相対パス形式のインクルードファイルにおける基点ディレクトリを変更
15	セクション名	section = | program = <セクション名> | const = <セクション名> | data = <セクション名> | bss = <セクション名> 省略時：(*p=P, c=C, d=D, b=B*)	 プログラム領域のセクション名を指定 定数領域のセクション名を指定 初期化データ領域のセクション名を指定 未初期化データ領域のセクション名を指定
16	コメントのネスト	comment	コメント(/* */)のネストを許す
17	文字列出力領域	string = *const* | data	文字列を定数領域セクション(C)へ出力 初期化データ領域セクション(D)へ出力
18	文字列内の日本語コードの選択	euc sjis	euc コードを選択 sjis コードを選択
19	漢字変換	outcode = euc | sjis	オブジェクト内漢字コードを euc コードにする オブジェクト内漢字コードを sjis コードにする
20	サブコマンドファイルの選択	subcommand = <ファイル名>	<ファイル名>で指定したファイルからコマンドオプションを取り込む
21	短絶対アドレスの指定	abs8 abs16	8 ビットデータを 8 ビット絶対アドレスでアクセス データを 16 ビット絶対アドレスでアクセス
22	列挙型サイズ	byteenum	enum 宣言した列挙型のデータを char 型で扱う
23	switch 文　展開方式	case = ifthen | table	インライン展開するかどうかを指定 インライン展開する関数のサイズの限界を指定
24	ブロック転送命令	eepmov	構造体の代入式を eepmov 命令で展開
25	メモリ間接形式	indirect	関数呼び出しをメモリ間接形式で展開
26	乗除算仕様の拡張解釈	cpuexpand *nocpuexpand*	乗除算を CPU 命令仕様に合わせてコード展開 乗除算を ANSI C 言語仕様準拠でコード展開
27	レジスタ変数の拡張	*regexpansion* noregexpansion	レジスタ変数に割り付けるレジスタ数を拡張する レジスタ変数に割り付けるレジスタ数を拡張しない
28	共通式の最適化	cmncode	共通式削除の最適化強化
29	外部変数の最適化	volatile *novolatile*	全ての外部変数を volatile 変数として最適化抑止 volatile 宣言のない外部変数を最適化

（1）CPU／動作モード（アドレス空間のビット幅）

■形式

cpu＝ 2600n｜
2600a［:〈アドレス空間のビット幅〉］｜
2000n｜
2000a［:〈アドレス空間のビット幅〉］｜
300hn｜
300ha［:〈アドレス空間のビット幅〉］｜
300｜300l｜300reg

■説明

作成するオブジェクトプログラムの CPU 種別と動作モードを指定します。サブオプションの一覧を**表** 1.3 に示します。

表 1.3　cpu **オプションのサブオプション一覧**

項番	サブオプション名	意　味
1	2600n	H8S/2600 **用ノーマルモードのオブジェクトを作成します。**
2	2600a［:〈アドレス空間のビット幅〉］	H8S/2600 **用アドバンストモードのオブジェクトを作成します。〈アドレス空間のビット幅〉は、**20、24、28、32 **のいずれかの数値で、それぞれ** 1M **バイト、**16M **バイト、**256M **バイト、**4G **バイトのアドレス空間を示します。〈アドレス空間のビット幅〉の省略時解釈は** 32 **です。**
3	2000n	H8S/2000 **用ノーマルモードのオブジェクトを作成します。**
4	2000a［:〈アドレス空間のビット幅〉］	H8S/2000 **用アドバンストモードのオブジェクトを作成します。〈アドレス空間のビット幅〉は、**20、24、28、32 **のいずれかの数値で、それぞれ** 1M **バイト、**16M **バイト、**256M **バイト、**4G **バイトのアドレス空間を示します。〈アドレス空間のビット幅〉の省略時解釈は** 32 **です。**
5	300hn	H8/300H **用ノーマルモードのオブジェクトを作成します。**
6	300ha［:〈アドレス空間のビット幅〉］	H8/300H **用アドバンストモードのオブジェクトを作成します。〈アドレス空間のビット幅〉は、**20 **または** 24 **の数値で、それぞれ** 1M **バイト、**16M **バイトのアドレス空間を示します。〈アドレス空間のビット幅〉の省略時解釈は** 24 **です。**
7	300	H8/300 **のオブジェクトを作成します。**
8	300l	H8/300 **のオブジェクトを作成します。クロスアセンブラとの互換のために用意しています。**
9	300reg	H8/300 **のオブジェクトを作成します。旧バージョンとの互換のために用意しています。**

■注意

（1）cpu オプションを省略した場合は、H38CPU 環境変数の内容を参照します。また、cpu オプションと H38CPU 環境変数を同時に指定した場合は、cpu オプションを優先します。

（2）cpu オプションと H38CPU 環境変数の両方を省略した場合はエラーとなります。

（2）最適化レベル

■形式

```
optimize＝0｜1
```

■説明

オブジェクトプログラムの最適化レベルを指定します。

optimize＝0 オプションは、C コンパイラがオブジェクトプログラムの最適化を行なわないことを示します。

optimize＝1 オプションは、C コンパイラが最適化を行なうことを示します。

本オプションの省略時解釈は、optimize＝1 とみなします。

■注意

optimize＝0 オプションを指定したとき、speed＝inline、loop オプションは無効となります。

（3）最適化内容

■形式

```
speed［=   register｜
           shift｜
           loop｜
           switch｜
           inline｜
           struct］
```

■説明

コンパイラが生成するオブジェクトに対し、実行速度の高速化を図る最適化を指定します。

speed＝register オプションは、CPU／動作モードが 300ha、300hn、300 の時、関数の入口／出口でレジスタを退避／回復するコードとして実行時ルーチンを使用せずに PUSH、POP 命令で展開します。

speed＝shift オプションは、シフト演算をより高速なオブジェクトコードで展開します。

speed＝loop、speed＝switch オプションは、for および switch 文のコード生成の高速化を指定します。

speed＝inline オプションは、呼び出す関数をインライン展開することを示します。インライン展開の条件については、**「2.3.1（4）関数のインライン展開」**を参照してください。

speed＝struct オプションは、構造体型や double 型の代入文を実行時ルーチンを用いず、直接インライン展開します。

speed のみを指定した場合は、これら全ての実行速度優先の最適化を行います。本オプションを省略した場合は、実行速度よりもオブジェクトコードのサイズ縮小を重視したオブジェクトを生成します。

■注意

最適化なし（optimize＝0）を指定したとき、speed＝loop, inline は無効となります。

（4）インライン展開

■形式

nestinline = <数値>

■説明

インライン展開対象関数の中でインライン展開対象関数を呼び出している場合、実際にインライン展開の対象とする、呼び出しネスト回数を指定します。指定できる最大数は 16 です。

また、オプション省略時には 1 を指定したものとして処理します。

■例

・ソースプログラム

```
#pragma inline(f1,f2,f3)
int a;
void f1(void){ a++; }
void f2(void){ f1(); }
void f3(void){ f2(); }
```

（a）nestinline＝1 のとき

```
#pragma inline(f1,f2,f3)
int a;
void f1(void){ a++; }
void f2(void){ a++; }
void f3(void){ f1(); }
```

（b）nestinline＝2 のとき

```
#pragma inline(f1,f2,f3)
int a;
void f1(void){ a++; }
void f2(void){ a++; }
void f3(void){ a++; }
```

（5）モジュール間最適化

■形式

goptimize

■説明

モジュール間最適化用付加情報の出力を指定します。

goptimize オプションが指定された場合は、オブジェクトプログラム出力ディレクトリに、「ch38iop」という名前のディレクトリを自動作成し、ディレクトリ「ch38iop」の下にソースプログラムと同じファイル名で拡張子「iop」の付加情報ファイルを出力します。goptimize オプションの指定がない場合は、付加情報は出力しません。モジュール間最適化の詳細は、**「5. モジュール間最適化」**を参照してください。

（6）デバッグ情報

■形式

debug

nodebug

■説明

debug オプションは、デバッグ時に C ソースプログラムレベルでのデバッグができるようにオブジェクトファイル中にデバッグ情報を出力することを指定します。

オブジェクトファイルがリロケータブルオブジェクトプログラムの時は、直接デバッグ情報が出力されます。また、アセンブリソースプログラムの時は、.LINE 制御命令がアセンブリソースプログラム中に組み込まれます。このため、C コンパイラが出力したアセンブリソースプログラムでも C ソースレベルのデバッグができます。

本オプションは、最適化オプションを指定した場合でも有効です。

nodebug オプションは、デバッグ情報をオブジェクトファイル中に出力しないことを指定します。

本オプションの省略時解釈は、nodebug です。

■注意

C ソースレベルのデバッグを行なうためには、アセンブルおよびリンク時にも debug オプションを指定する必要があります。

（7）リスト内容と形式

■形式

show＝ source｜nosource｜
object｜noobject｜
statistics｜nostatistics｜
allocation｜noallocation｜
expansion｜noexpansion
width＝〈数値〉｜
length＝〈数値〉

■説明

コンパイラが出力するリストの内容とその形式、および出力の解除を指定します。サブオプションの一覧を**表** 1.4 に示します。

表 1.4　show **オプションのサブオプション一覧**

項番	サブオプション名	意　味
1	source	**ソースプログラムのリストを出力します。**
2	nosource	**ソースプログラムのリストを出力しません。**
3	object	**オブジェクトプログラムのリストを出力します。**
4	noobject	**オブジェクトプログラムのリストを出力しません。**
5	statistics	**統計情報のリストを出力します。**
6	nostatistics	**統計情報のリストを出力しません。**
7	allocation	**シンボル割り付け情報のリストを出力します。**
8	noallocation	**シンボル割り付け情報のリストを出力しません。**
9	expansion	**インクルードファイル、マクロ展開した後のソースプログラムリストを出力します。** nosource **サブオプションが同時に指定された場合には、**expansion **サブオプションは無効となり、ソースプログラムリストは出力されません。**
10	noexpansion	**インクルードファイル、マクロを展開する前のソースプログラムリストを出力します。** nosource **サブオプションが同時に指定された場合には、**expansion **サブオプションは無効となり、ソースプログラムリストは出力されません。**

項番	サブオプション名	意　　味
11	width＝〈数値〉	〈数値〉で指定する文字数をリストの 1 行の最大文字数とします。 〈数値〉は 10 進数で指定し、0、または 80 から 132 の間の数値を指定することができます。 〈数値〉が 0 の場合、リストの 1 行の最大文字数は規定されません。
12	length＝〈数値〉	〈数値〉で指定する行数を、リストの 1 ページの最大行数とします。 〈数値〉は 10 進数で指定し、0、または 20 から 255 の間の数値を指定することができます。 〈数値〉が 0 の場合、リストの 1 ページの最大行数は規定されません。

本項で記した各リストの具体例については**「1.3.6　コンパイルリストの見方」**を参照してください。

本オプションの省略時解釈は、

show＝source, noobject, statistics, noallocation, noexpansion, width＝0, length＝0

です。

（8）リストファイル

■形式

list＝［＝〈リストファイル名〉］

nolist

■説明

list オプションは、〈リストファイル名〉を指定することができます。

nolist オプションを指定すると、リストファイルは作成されません。

〈リストファイル名〉は、**「1.3.2　ファイル名の付け方」**に従って指定できます。

list オプションで〈リストファイル名〉を指定しない場合には、ソースファイルと同じファイル名で、拡張子が UNIX 版のとき「lis」、MS-DOS 版のとき「lst」のリストファイルが作成されます。

本オプションの省略時解釈は list です。

(9) インフォメーションメッセージ出力

■形式

message

■説明

message オプションは、C ソースプログラムのインフォメーションレベルのメッセージを出力することを指定します。

message オプションを省略した場合は、インフォメーションレベルのメッセージは出力されません。

(10) オブジェクトファイル

■形式

object［＝〈オブジェクトファイル名〉］

noobject

■説明

object オプションは、〈オブジェクトファイル名〉を指定することができます。

noobject オプションを指定すると、オブジェクトファイルは作成されません。

object オプションで〈オブジェクトファイル名〉指定をしない場合には、ソースファイルと同じファイル名で拡張子が「obj」（出力ファイルがリロケータブルオブジェクトプログラムの時）、または「src」（出力ファイルがアセンブリソースプログラムの時）のオブジェクトファイルが作成されます。

ファイル拡張子が「obj」か「src」かは、code オプションで決まります。

本オプションの省略時解釈は object です。

■注意

noobject オプションを指定したとき、以下のオプション指定は無効となります。

case,
code,
cpuexpand/nocpuexpand,
debug,
eepmov,
optimize,
cmncode,
regexpansion/noregexpansion,

```
section,
show=object, statistics, allocation,
speed,
string,
volatile/novolatile
```

（11）オブジェクト形式

■形式

code＝machinecode｜asmcode

■説明

オブジェクトプログラムの種類を指定します。

code＝machinecode オプションを指定すると、リンケージエディタに入力できるリロケータブルオブジェクトプログラムが作成されます。

code＝asmcode オプションを指定すると、アセンブラに入力できるアセンブリソースプログラムが作成されます。

code＝asmcode と debug オプションを同時に指定すると、アセンブリソースプログラム中に.LINE 制御命令が出力されます。アセンブル時にも、debug オプションを指定することにより、C ソースプログラムレベルでのデバッグが可能です。

本オプションを省略した場合は、code＝machinecode が指定されたものと解釈します。

■注意

code＝asmcode オプションを指定したとき、show＝object は無効となります。

（12）マクロ名の定義

■形式

define＝　〈マクロ名〉＝〈名前〉｜
　　〈マクロ名〉＝〈定数〉｜
　　〈マクロ名〉

■説明

マクロ名を定義します。

サブオプションで指定できるマクロ名、名前および定数の定義を**表** 1.5 に示します。マクロ名、名前および定数の長さはそれぞれ先頭から 31 文字までが有効となります。サブオプションの〈マクロ名〉＝〈名前〉および〈マクロ名〉＝〈定数〉の形式では、それぞれ名前または定数をマクロ名として定義できます。

サブオプションに〈マクロ名〉を単独で指定した場合は、そのマクロ名が定義されたものと仮定することができます。

本オプション指定を用いれば、C ソースプログラム中と同様のマクロ定義がコマンドラインのオプション指定でできます。

サブオプションは 16 個まで指定できます。

表 1.5　define オプションで指定できるマクロ名、名前、定数

No.	項　目	説　　明
1	**マクロ名**	**英字またはアンダラインで始まり、そのあとに 0 個以上の英字、アンダラインまたは数字が続く文字列です。**
2	**名　前**	**英字またはアンダラインで始まり、そのあとに 0 個以上の英字、アンダラインまたは数字が続く文字列です。**
3	**定　数**	10 **進定数：1 個以上の数字（0〜9）の繰り返し、または 1 個以上の数字の繰り返しにピリオドが続き、そのあとに 0 個以上の数字が続く文字列です。** 8 **進定数：数字０で始まり、その後に１個以上の０〜７の数字が続く文字列です。** 16 **進定数：数字０に x が続き、その後に１個以上の数字、または A〜F, a〜f が続く文字列です。**

（13）インクルードファイル

■形式

include＝〈パス名〉

■説明

include オプションは、コンパイルする C ソースプログラムが参照するインクルードファイルの存在するパス名を指定します。

パス名が複数ある場合にはカンマ（, ）で区切って指定することができます。

インクルードファイルの検索方法については、**「付録 A.1（13）プリプロセッサの仕様」**を参照してください。

（14）インクルードファイル基点ディレクトリ

■形式

chgincpath

■説明

ディレクトリ相対形式で指定されたインクルードファイルを検索するとき、ソースファイルのあるディレクトリを基点のディレクトリとします。指定のない場合は、コンパイラを起動したカレントディレクトリを基点ディレクトリとします。

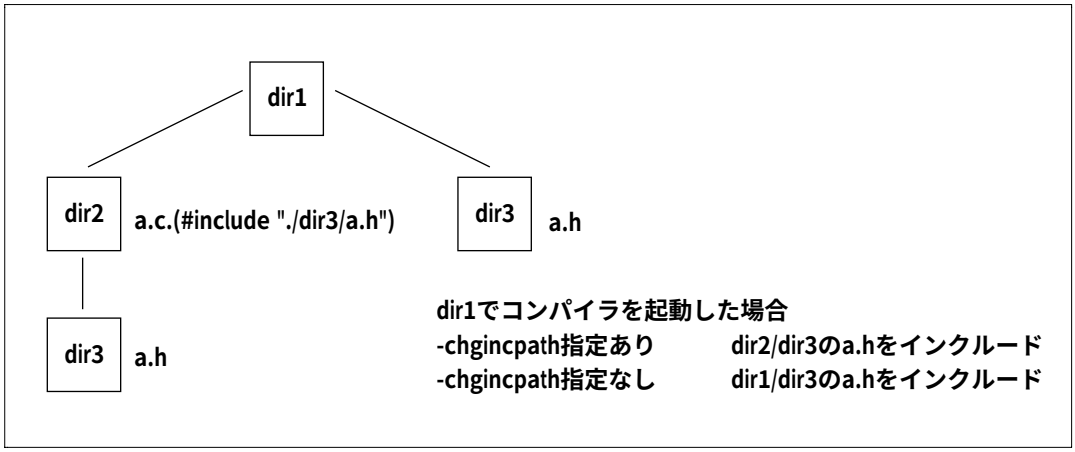


図 1.2 **基点ディレクトリ**

（15）セクション名

■形式

section＝ program＝〈セクション名〉｜
const＝〈セクション名〉｜
data＝〈セクション名〉｜
bss＝〈セクション名〉

■説明

オブジェクトプログラム中のセクション名を指定します。

セクション名についての詳細は、**「**2.2.1 **オブジェクトプログラムの構造」**を参照してください。〈セクション名〉は、英字、数字、アンダーライン（_）または、$の列で、先頭が数字以外のものです。セクション名は、32 文字目まで有効です。

本オプションの省略時解釈は、section＝program＝P, const＝C, data＝D, bss＝B です。

（16）コメントのネスト

■形式

com**m**ent

■説明

ネストしたコメントの記述を可能にします。

本オプションを省略した場合、コメントのネストを記述するとエラーになります。

■例

```
/* This is an example of/* nested */ comment */
                                    ↑
                                   [1]
```

comment オプションを指定すると全てコメントと解釈しますが、省略した場合は［1］でコメントが終わっていると解釈します。

（17）文字列出力領域

■形式

string＝**c**onst｜**d**ata

■説明

string＝const オプションは、C ソースプログラムの文字列を定数領域に出力することを指定します。

定数領域に出力した文字列はプログラム実行時に RAM へ転送する必要がありません。

string＝data オプションは、文字列を初期化データ領域に出力することを指定します。初期化データ領域へ出力した文字列はプログラム実行時に変更することができます。

本オプションの省略時解釈は、string＝const です。

（18）文字列内の日本語コードの選択

■形式

euc｜**s**jis

■説明

euc オプションは、文字列、文字定数およびコメント内に記述した日本語を EUC コードとして扱うことを指定します。

sjis オプションは、文字列、文字定数およびコメント内に記述した日本語をシフト JIS

コードとして扱うことを指定します。

本オプションを省略した場合は、コメント内の日本語のみホストマシンの日本語環境コードとして認識し、文字列、文字定数に記述した日本語は不定コードとして扱います。

（19）漢字変換

■形式

```
outcode = euc | sjis
```

■説明

文字列、文字定数内に日本語を記述したときに、オブジェクトプログラムに出力する漢字コードを指定することができます。

outcode = euc 指定時は、漢字コードを euc コードで出力します。

outcode = sjis 指定時は、漢字コードを sjis コードで出力します。

ソースプログラム上の漢字コードは、euc または sjis オプションで指定できます。

（20）サブコマンドファイルの選択

■形式

```
subcommand＝〈サブコマンドファイル名〉
```

■説明

subcommand オプションは、C コンパイラ起動時のコンパイラオプションをサブコマンドファイルで指定します。サブコマンドファイル中の書式は、コマンドラインの書式と同一です。

■例

サブコマンドファイル（opt.sub）の内容が

```
-show＝object, length＝0 -debug -byteenum
```

のとき、

```
ch38 -cpu＝2600a -subcommand＝opt.sub test.c
```

の指定は

```
ch38 -cpu＝2600a -show＝object, length＝0 -debug -byteenum test.c
```

としてコンパイルします。

（21）短絶対アドレスの指定

■形式

```
abs8
abs16
```

■説明

静的領域に割り付けるデータを、短絶対アドレッシングモードでアクセスします。

abs8 オプションは、char 型、unsigned char 型および char 型、unsigned char 型の要素、メンバを含む複合型データを 8 ビット絶対アドレス（@aa:8）でアクセスするコードを生成します。

abs16 オプションは、CPU／動作モードが 2600a、2000a、300ha のとき、データを 16 ビット絶対アドレス（@aa:16）でアクセスするコードを生成します。CPU／動作モードが 2600n、2000n、300hn、300 のとき、本オプションの指定は無効です。

abs8 オプションにより、8 ビット絶対アドレスでアクセスされるデータは、セクション名“ $ABS8+C セクション名 ”、“ $ABS8+D セクション名 ”または“ $ABS8+B セクション名 ”に出力されます。また、abs16 オプションにより、16 ビット絶対アドレスでアクセスされるデータは、セクション名“ $ABS16+C セクション名 ”、“ $ABS16+D セクション名 ”または“ $ABS16+B セクション名 ”に出力されます。リンク時には、本オプションにより出力されたセクションを短絶対アドレス領域に割り付ける必要があります。

短絶対アドレス領域の範囲については、**「付録Ｆ　短絶対アドレスのアクセス範囲」**を参照してください。また、短絶対アドレス領域のセクション名の指定方法については、「2.3.1（7）**セクション切り替え機能**」を参照してください。

（22）列挙型サイズ

■形式

```
byteenum
```

■説明

enum 宣言した列挙型のデータを char 型として扱います。

本オプションが指定された場合で、かつ、enum 宣言した列挙型のメンバの値が全て-128～127 の範囲のとき、列挙型データを char 型として扱います。

本オプションを省略した場合、および、本オプションが指定されても列挙型のメンバの値が 1 つでも-128～127 の範囲外の場合は、列挙型データを int 型として扱います。

（23）switch 文展開方式

■形式

case=ifthen｜table

■説明

switch 文のコード展開方式を指定します。

case=ifthen オプションは、switch 文を if_then 方式で展開します。if_then 方式は、switch 文の評価式の値と case ラベルの値を比較し、一致すれば case ラベルの文へ飛ぶ処理を case ラベルの回数繰り返す展開方式です。この展開方式は、switch 文に含まれる case ラベルの数に比例してオブジェクトコードのサイズが増大します。

case=table オプションは、switch 文をテーブル方式で展開します。テーブル方式は、case ラベルの飛び先をジャンプテーブルに確保し、1 回のジャンプテーブルの参照で switch 文の評価式と一致する case ラベルの文へ飛び越す展開方式です。この方式は、switch 文に含まれる case ラベルの数に比例して定数領域に確保されるジャンプテーブルのサイズが増えますが、実行速度は常に一定です。

本オプションを省略した場合は、オブジェクトサイズの縮小を優先したいずれかの展開方式を C コンパイラが自動的に選択します。

また、本オプション省略時に speed オプションあるいは speed＝switch オプションを指定した場合は、実行速度を優先した展開方式を C コンパイラが自動的に選択します。

■例

```
int a, b;
   :
switch(a){
      case 1:       b=3; break;
      case 2:       b=2; break;
      case 3:       b=1; break;
      default:      b=0; break;
}
```

上記の C ソースプログラムのコード展開例を次に示します。（cpu＝2600n の場合）

```
MOV.W   @_a,R0          MOV.W   @_a,R0
MOV.B   R0H,R0H         SUB.W   #1,R0
BNE     Ld              CMP.W   #2,R0
CMP.B   #1,R0L          BHI     Ld
BEQ     L1              ADD.W   R0,R0
CMP.B   #2,R0L          MOV.W   @(L,ER0),R0
BEQ     L2              JMP     @ER0
CMP.B   #3,ROL          ：
BEQ     L3              L:（ジャンプテーブル）
BRA     Ld
case=ifthen 時          case=table 時
```

表 1.6　switch 文　展開方式の比較

case 値	if_then 方式		テーブル方式	
	オブジェクトサイズ	実行サイクル	オブジェクトサイズ	実行サイクル
1	22 バイト	18	28（22+6）バイト	28
3		30		

（24）ブロック転送命令

■形式

```
eepmov
```

■説明

構造体の代入文や局所変数で宣言された配列の初期値代入式を、ブロック転送命令 EEPMOV にコード展開します。

本オプションを省略した場合は、構造体の代入文などを MOV 命令または、実行時ルーチンに展開します。

■注意

EEPMOV 命令実行中に NMI 割り込みが発生すると、割り込み処理終了後、次の命令に制御が移るため動作結果が保証されません。本オプション使用時には NMI 割り込みに注意してください。

（25）メモリ間接形式

■形式

```
indirect
```

■説明

C ソースプログラム内で呼び出す関数を全てメモリ間接（@@aa:8）で呼び出します。

本オプションの指定により、C ソースプログラム中に定義されている関数は、関数本体以外にセクション名 “ $INDIRECT+セクション名 ” にメモリ間接呼び出しのためのアドレステーブルが出力されます。アドレステーブルのセクション名の指定方法については、**「**2.3.1（7）**セクション切り替え機能」**を参照してください。

■注意

アドレステーブルを格納できるエリアは、0x0000～0x00FF 番地に制限されています。リンク時には、リンケージマップによりアドレステーブルのセクションが 0x0000～0x00FF 番地の範囲内であることを確認してください。

（26）乗除算仕様の拡張解釈

■形式

cpuexpand

nocpuexpand

■説明

変数の乗除算のコード展開を ANSI の C 言語規格から拡張解釈して生成します。

nocpuexpand オプションは、乗除算のコード展開を ANSI の C 言語規格に準拠した形で生成します。本オプションの指定による乗除算のコード展開を**表** 1.7 に示します。

表 1.7　cpuexpand **オプションの演算仕様**

対象演算	us1*us2 **の演算サイズ（**H8S/2600 **の例）**	
	cpuexpand **指定時**	nocpuexpand **指定時**
unsigned 　short us1, us2; unsigned long ul; 　ul＝us1*us2;	us1*us2 **は** unsigned long **で演算します。** **出力例：** MOV.W @_us1,Rd MOV.W @_us2,Rs MULXU.W Rs,ERd MOV.L ERd,@_ul us1*us2 **の結果** 4 **バイトを** u1 **に代入します。**	us1*us2 **は** unsigned short **で演算します。** **出力例：** MOV.W @_us1,Rd MOV.W @_us2,Rs MULXU.W Rs,ERd EXTU.L ERd MOV.L ERd,@_ul us1*us2 **の結果の下位** 2 **バイトを** 0 **拡張して** ul **に代入します。**
unsigned 　short us1, us2, us3; unsigned short us; 　us＝us1*us2/us3;	us1*us2 **は** unsigned long **で演算します。** **出力例：** MOV.W @_us1,Rd MOV.W @_us2,Rs MULXU.W Rs,ERd MOV.W @_us3,Rs DIVXU.W Rs,ERd MOV.W Rd,@_us us1*us2 **の結果** 4 **バイトを除算命令の被除数にします。**	us1*us2 **は** unsigned short **で演算します。** **出力例：** MOV.W @_us1,Rd MOV.W @_us2,Rs MULXU.W Rs,ERd EXTU.L ERd MOV.W @_us3, Rs DIVXU.W Rs,ERd MOV.W Rd,@_us us1*us2 **の結果の下位** 2 **バイトを** 0 **拡張した値を除算命令の被除数にします。**

■注意

cpuexpand オプションを指定した場合、言語仕様で規定された値の保証範囲と仕様が異なるため、演算結果が nocpuexpand オプション指定時と異なる場合があります。

（27）レジスタ変数の拡張

■形式

regexpansion
noregexpansion

■説明

regexpansion オプションは、レジスタ変数を割り付けるレジスタの数を拡張することを指定します。

noregexpansion オプションは、レジスタ変数を割り付けるレジスタの数の拡張を行わないことを指定します。

レジスタの数を拡張した場合、一般にレジスタに割り付く変数の数が多くなり、変数のアクセススピードが速くなります。

レジスタ変数の割り付け規則については、**「付録 A.1（8）レジスタ」**を参照してください。

本オプションの省略時解釈は、regexpansion です。

（28）共通式の最適化

■形式

cmncode

■説明

共通式をテンポラリ変数に変換する最適化で、対象となる式の数を拡張します。

cmncode オプション指定により共通式最適化の対象式を拡張すると、テンポラリ変数をレジスタに割り付け、一般的にはオブジェクト性能がよくなります。しかし、レジスタの数が不足するとテンポラリ変数がメモリに割り付いて、逆にオブジェクト性能が低下してしまうことがあります。本オプションは、プログラムによってオブジェクト性能向上の効果が変わりますので、性能チューニング時に試してみてください。

（29）外部変数の最適化

■形式

volatile｜novolatile

■説明

volatile オプションは、外部変数に対して最適化を行いません。

novolatile オプションは、外部変数に対して最適化を行います。

本オプションの省略時解釈は、novolatile です。

1.3.4 コンパイラの環境変数

コンパイラで使用する環境変数の使用方法を**表** 1.8 に示します。

表 1.8 **環境変数**

No.	環境変数	説明
1	path	**コンパイラ本体の格納ディレクトリを指定します。** **指定フォーマット：** PC **版** A> path= <**コンパイラ本体パス名**>;[<**既存パス名**>;...] unix C **シェル** %set path =(<**コンパイラ本体パス名**> $path) **ボーンシェル** %PATH=:<**コンパイラ本体パス名**>[:<**既存パス名**>...] %export PATH
2	H38CPU	**コンパイラの** -cpu **オプションによる** CPU **種別の指定を、環境変数によって指定します。以下の指定が可能です。** CPU／**動作モード** **アドレス空間のビット幅** **省略時解釈** 2600n ― ― 2600a 20 | 24 | 28 | 32 32 2000n ― ― 2000a 20 | 24 | 28 | 32 32 300hn ― ― 300ha 20 | 24 24 300 ― ― 300l ― ― **上記以外の指定はエラーとなります。** H38CPU **環境変数による** CPU **の指定と、**-cpu **オプションによる** CPU **の指定が相反する場合は、ウォーニングメッセージを出力し、**-cpu **オプションの指定を優先します。** **指定フォーマット：** PC **版** A> set H38CPU= <CPU／**動作モード**>[:<**アドレス空間のビット幅**>] unix C **シェル** %setenv H38CPU <CPU／**動作モード**>[:<**アドレス空間のビット幅**>] **ボーンシェル** %H38CPU=<CPU／**動作モード**>[:<**アドレス空間のビット幅**>] %export H38CPU

No.	環境変数	説明
3	CH38	インクルードファイル格納ディレクトリを指定します。 システムインクルードファイルの検索順序は、include オプション指定ディレクトリ、CH38 指定ディレクトリとなります。 ユーザインクルードの検索順序は、カレントディレクトリ、include オプション指定ディレクトリ、CH38 指定ディレクトリとなります。 環境変数 CH38 の指定がない場合、UNIX 版では/usr/CH38 を仮定します。PC 版には省略時解釈はありません。 指定フォーマット： PC 版　A> set CH38= <インクルードパス名> unix　C シェル　%setenv CH38 <インクルードパス名> ボーンシェル　% CH38 = <インクルードパス名> %export　CH38
4	CH38TMP	コンパイラがテンポラリファイルを作成するディレクトリを指定します。この環境変数の指定がない場合は、カレントディレクトリにテンポラリファイルを作成します。 指定フォーマット： PC 版　A> set CH38TMP= <テンポラリファイルパス名> unix　C シェル　%setenv CH38TMP <テンポラリファイルパス名> ボーンシェル　% CH38TMP = <テンポラリファイルパス名> %export　CH38TMP

1.3.5 標準ライブラリとの対応

標準ライブラリとして、コードサイズ優先／スピード優先を選択できます。標準ライブラリには、次の 10 種類があります。リンク時に、cpu オプションおよびコードサイズ優先／スピード優先の選択により**表** 1.9 に示すライブラリを指定してください。

c38reg.lib、c38regs.lib （H8/300 用）
c38hn.lib、c38hns.lib （H8/300H ノーマルモード用）
c38ha.lib、c38has.lib （H8/300H アドバンストモード用）
c8s26n.lib、c8s26ns.lib （H8S/2000,H8S/2600 ノーマルモード用）
c8s26a.lib、c8s26as.lib （H8S/2000,H8S/2600 アドバンストモード用）

表 1.9 **標準ライブラリとコンパイルオプションの関係**

cpu/**動作モード**	**オプション**	**コードサイズ／スピード選択**	
		コードサイズ優先	**スピード優先**
H8/300	cpu=300	c38reg.lib	c38regs.lib
H8/300H **ノーマルモード**	cpu=300hn	c38hn.lib	c38hns.lib
H8/300H **アドバンストモード**	cpu=300ha	c38ha.lib	c38has.lib
H8S/2000,H8S/2600 **ノーマルモード**	cpu=2000n cpu=2600n	c8s26n.lib	c8s26ns.lib
H8S/2000,H8S/2600 **アドバンストモード**	cpu=2000a cpu=2600a	c8s26a.lib	c8s26as.lib

■注意

ライブラリのコードサイズ優先、スピード優先の選択は、コンパイラオプションの speed とは関係なく指定することができます。

1.3.6 コンパイルリストの見方

本節では、コンパイルリストの内容と形式について説明します。

(1) コンパイルリストの構成

コンパイルリストの構成と内容を**表** 1.10 に示します。

表 1.10 **コンパイルリストの構成と内容**

No.	リストの作成	内　　容	オプション指定方法	オプション省略時
1	ソースリスト情報	ソースプログラムのリスト*1	show＝source show＝nosource	出力する
		インクルードファイル、マクロ展開後のソースプログラムのリスト*2	show＝expansion show＝noexpansion	出力しない
2	エラー情報	コンパイル時のエラー情報	—	出力する
3	シンボル割り付け情報	関数のスタックフレームでの変数割り付け情報	show＝allocation show＝noallocation	出力しない
4	オブジェクト情報	オブジェクトプログラムの機械語、アセンブリコード	show＝object show＝noobject	出力しない
5	統計情報	各セクションのバイト数、シンボル数情報、オブジェクト種類	show＝statistics show＝nostatistics	出力する

【注】 *1 **ソースプログラムのリストは、サブオプションの組み合わせにより、オブジェクト情報内に出力される場合もあります。サブオプションの組み合わせによるソースプログラムリストの出力先を表** 1.11 **に示します。**

*2 **インクルードファイル、マクロ展開後のソースプログラムのリストは** show＝source **指定時に有効になります。**

表 1.11 **ソースプログラムリストの出力先**

ソースプログラムリストの出力先	サブオプションの指定		
	source	expansion	object
ソースリスト情報として出力	指定あり	指定あり	−
オブジェクト情報として出力	指定あり	指定なし	指定あり

(2) ソースリスト情報

ソースリスト情報の出力形式には、プリプロセッサを通すまえのソースプログラムを出力する形式（show＝noexpansion を指定する場合）と、プリプロセッサを通したあとのソースプログラムを出力する形式（show=expansion を指定する場合）があります。それぞれ

の出力形式を**図** 1.3（a）、（b）に示します。また、**図** 1.3（b）に相違点を網掛けで示します。

（a）show=noexpansionのソースリスト情報

```
************ SOURCE LISTING ************

FILE NAME: m0260.c

   Seq File      Line Pi 0----+----1----+----2----+----3----+----4----+----5---
     1 m0260.c      1     #include "header.h"
     4 m0260.c      2
     7 m0260.c      5     int sum2(void)
     8 m0260.c      6     {   int j;
    10 m0260.c      8
    11 m0260.c      9     #ifdef SMALL
    12 m0260.c     10         j=SML_INT;
    13 m0260.c     11     #else
    14 m0260.c     12         j=LRG_INT;
    15 m0260.c     13     #endif
    16 m0260.c     14
    17 m0260.c     15         return j; /* continue12345678901234567890123456
    [1]  [2]       [3]    +2345678901234567890  */
    18 m0260.c     16    [6]
                             }
```

（b）show=expansionのソースリスト情報

```
************ SOURCE LISTING ************

FILE NAME: m0260.c

  Seq File        Line Pi 0----+----1----+----2----+----3----+----4----+----5---
    1 m0260.c        1     #include "header.h"
    2 header.h       1     #define SML_INT         1
    3 header.h       2     #define LRG_INT       100
    4 m0260.c        2
    7 m0260.c        5     int sum2(void)
    8 m0260.c        6     {   int j;
   10 m0260.c        8
   11 m0260.c        9     #ifdef SMALL
   12 m0260.c       10 X       j=SML_INT;
   13 m0260.c       11[4]  #else
   14 m0260.c       12  E      j=100;
   15 m0260.c       13 [5] #endif
   16 m0260.c       14
   17 m0260.c       15         return j; /* continue12345678901234567890123456
   [1]  [2]         [3]    +2345678901234567890  */
   18 m0260.c       16    [6]
                            }
```

【注】

[1] リスト上の行番号
[2] ソースプログラムファイル名、またはインクルードファイル名
[3] ソースプログラムまたはインクルードファイル内の行番号
[4] show＝expansion**指定時、＃**ifdef**文、＃**elif**文等の条件コンパイル文でコンパイル対象とならないソース行**
[5] show＝expansion**指定時、＃**define**文によるマクロ置換のあったソース行**
[6] ソースプログラムの一行がコンパイルリストの一行に入りきらず、複数行にまたがって表示されたソース行

図 1.3　**ソースリスト情報の出力形式**

(3) エラー情報

エラー情報の出力例を**図** 1.4 に示します。

```
************ SOURCE LISTING ************

FILE NAME: m0260.c

  Seq File        Line Pi 0----+----1----+----2----+----3----+----4----+----5---
    1 m0260.c        1     #include "header.h"
    4 m0260.c        2
    5 m0260.c        3     extern int sum3(int);
    6 m0260.c        4
    7 m0260.c        5     sum3(int x)
    8 m0260.c        6     {
    9 m0260.c        7         int i;
   10 m0260.c        8         int j;
   11 m0260.c        9
   12 m0260.c       10         j=0;
   13 m0260.c       11         for (i=0; i<=x; i++){
   14 m0260.c       12             j+=k;   <-------- エラー発生行
   15 m0260.c       13         }
   16 m0260.c       14         return j;
   17 m0260.c       15      }
   18 m0260.c       16
   19 m0260.c       17
   20 m0260.c       18



  *********** ERROR INFORMATION **********

FILE NAME: m0260.c


File        Line Erno Lvl  Message
m0260.c       12 2225 (E)  UNDECLARED NAME: "k"
  [1]         [2] [3] [4]      [5]

NUMBER OF ERRORS:          1 }[6]
NUMBER OF WARNINGS:        0
NUMBER OF INFORMATIONS     0  [7]
```

【注】
[1] エラーの発生したソースプログラム名，先頭から10文字まで表示
[2] エラーの発生したソースプログラム中の行番号
[3] エラーメッセージを識別するための番号
[4] (I) インフォメーションレベル
(W) ウォーニングレベル
(E) エラーレベル
(F) フェータルレベル
[5] エラーの内容
[6] エラーレベルメッセージ，ウォーニングレベルメッセージの総数
[7] インフォメーションレベルメッセージの総数
(messageオプションを指定した時のみ)

図 1.4 **エラーを含んだソースエラーリストとエラー情報**

（4）シンボル割り付け情報

関数の引数や局所変数の割り付け情報を表します。H8S/2600 用アドバンストモードでコンパイルしたときのシンボル割り付け情報の例を**図** 1.5 に示します。

```
************ SOURCE LISTING ************

FILE NAME: m0280.c

Seq File      Line Pi 0----+----1----+----2----+----3----+----4----+----5---((
  1 m0280.c      1    extern int h(char, char *, double );                    ))
  2 m0280.c      2
  3 m0280.c      3    int
  4 m0280.c      4    h(char a, register char *b, double c)
  5 m0280.c      5    {
  6 m0280.c      6       char    *d;
  7 m0280.c      7
  8 m0280.c      8       d= &a;
  9 m0280.c      9       h(*d,b,c);
 10 m0280.c     10       {
 11 m0280.c     11            register int i;
 12 m0280.c     12
 13 m0280.c     13            i= *d;
 14 m0280.c     14            return i;
 15 m0280.c     15       }
 16 m0280.c     16    }

******* STACK FRAME INFORMATION ********

FILE NAME: m0280.c

Function (File m0280.c   , Line   4): h
                  [1]
 Parameter Allocation
   a                                  0xfffffff7 saved from R0L
   b                              REG ER5        saved from ER1 } [2]
   c                                  0x00000008                 )

 Level  1 (File m0280.c   , Line    5) Automatic/Register Variable Allocation
   d                                  0xfffffff2                             }[3]
 Level  2 (File m0280.c   , Line   10) Automatic/Register Variable Allocation
   i                              REG R4                                     )

Parameter Area Size     : 0x00000008 Byte(s)
Linkage Area Size       : 0x00000008 Byte(s)
Local Variable Size     : 0x00000006 Byte(s)  } [4]
Temporary Size          : 0x00000000 Byte(s)
Register Save Area Size : 0x00000008 Byte(s)
Total Frame Size        : 0x0000001e Byte(s)
```

【注】

［1］関数が定義されたファイル名，行番号，関数名

［2］引数の割り付け場所 A saved from B　**Bで渡された引数を関数入口でAにコピーした場合**
REG ERx　**割り付け場所がレジスタの場合，**REG**を表示**
0xffffffxx　**割り付け場所がスタックの場合，フレームポインタ（**ER6**）からのオフセットを表示**

［3］複文内で宣言された局所変数の割り付け場所，スタックの場合はER6**からのオフセット，レジスタの場合は**REG**を表示**

［4］関数内で使用するスタックフレームの割り付け情報

Parameter Area Size	**：スタックで渡される引数領域とリターン値アドレス領域のサイズ**
Linkage Area Size	**：リンケージ領域 (リターンアドレス領域＋フレームポインタ退避領域)の合計サイズ**
Local Variable Size	**：関数内で使用する局所変数領域とレジスタで渡された引数がスタックに割り付けられた場合に使用する引数退避領域の合計サイズ**
Temporary Size	**：関数内でＣコンパイラが使用するテンポラリ領域のサイズ**
Register Save Area Size	**：関数内で使用するレジスタの値を退避しておく領域のサイズ**
Total Frame Size	**：関数内で割り付けるスタックフレームの合計サイズ**

図 1.5　**シンボル割り付け情報**（cpu=2600a）

■注意

最適化オプション optimize＝1 が指定されていると引数割り付け情報および局所変数割り付け情報は出力しません。このとき以下のメッセージを出力します。

```
Optimize Option Specified : No Allocation Information Available
```

図 1.5 のシンボル割り付け情報に対応する、スタック上の割り付け例を**図** 1.6 **に**示します。

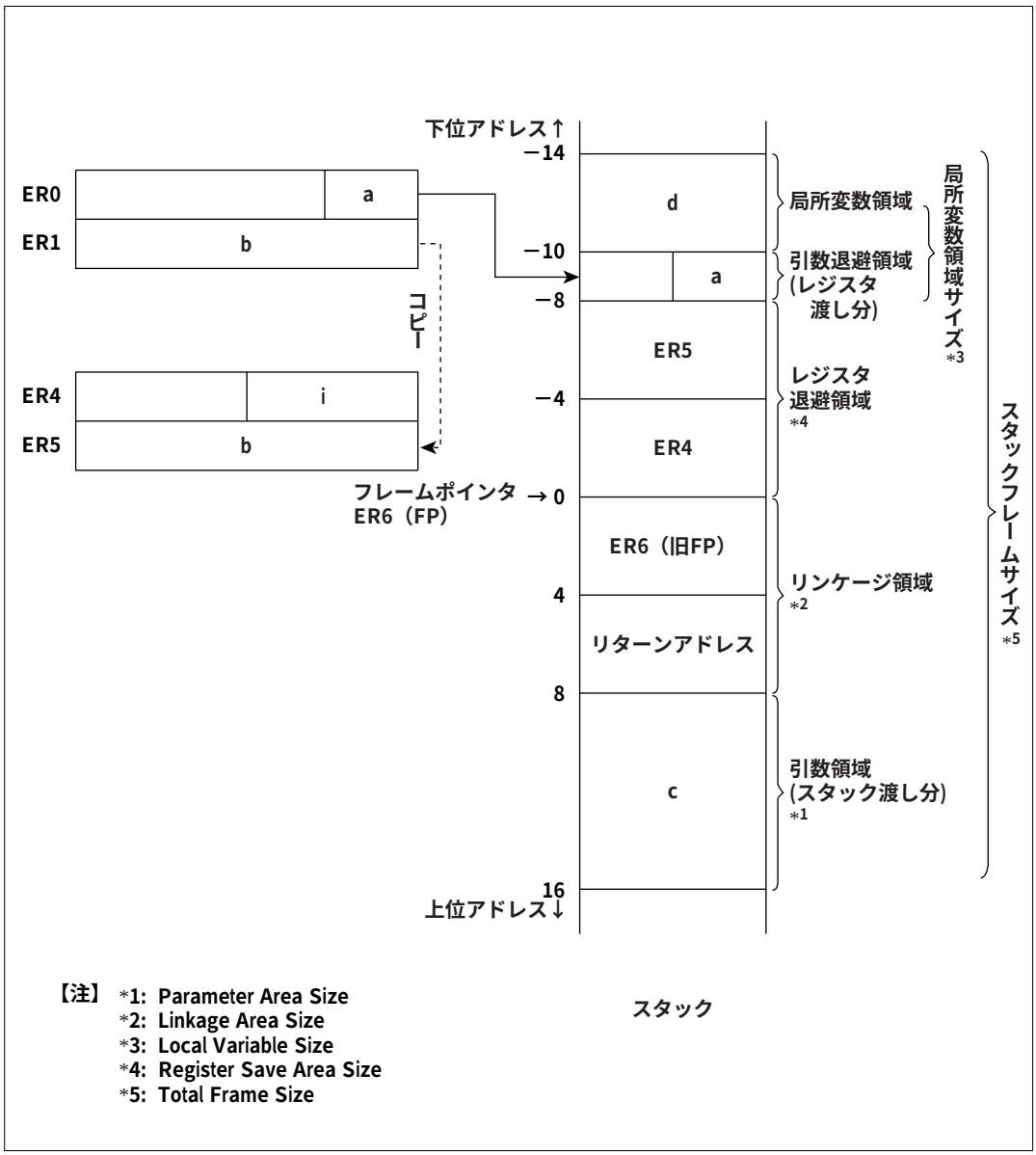


図 1.6　**スタック上の割り付け例**（cpu=2600a）

(5) オブジェクト情報

オブジェクト情報にソースプログラムのリストが出力される場合と、出力されない場合のリスト例を**図** 1.7、**図** 1.8 に示します。

```
************ OBJECT LISTING ************

FILE NAME: m0251.c

SCT OFFSET CODE        C LABEL    INSTRUCTION OPERAND   COMMENT
[1]  [2]  [3]               [4]
P                                                      ; section
                    1:    extern int sum(int);
                                 [5]
                    2:
                    3:    int
                    4:    sum(int x)
  00000000            _sum:                            ; function: sum
                    5:    {
                    6:    int i;
                    7:    int j;
                    8:
                    9:    j=0;
  00000000 1911               SUB.W       R1,R1
                   10:    for (i=0; i<=x; i++){
  00000002 1988               SUB.W       E0,E0
  00000004 4004               BRA         L8:8
  00000006              L7:
                   11:      j+=i;
  00000006 0981               ADD.W       E0,R1
  00000008 0B58               INC.W       #1,E0
  0000000A              L8:
  0000000A 1D08               CMP.W       R0,E0
  0000000C 4FF8               BLE         L7:8
                   12:        }
                   13:        return j;
  0000000E 0D10               MOV.W       R1,R0
                   14:    }
  00000010 5470               RTS
```

【注】
[1] 各セクションのセクション属性（P，C，D，B）
[2] 各セクションの先頭からのオフセット
[3] 各セクションのオフセットアドレスの内容
[4] 機械語に対応するアセンブリコード
[5] ソースプログラム内の行番号とソースリスト

図 1.7 **ソースプログラムリストが出力される場合のオブジェクト情報**
（show＝source, object, cpu＝2600a）

■注意

show＝expansion オプションを指定した場合は、常に**図** 1.8 のオブジェクト情報となります。

```
************ OBJECT LISTING ************

FILE NAME: m0251.c

SCT OFFSET CODE       C LABEL     INSTRUCTION OPERAND    COMMENT
[1]   [2]   [3]                       [4]
P                                                        ; section
                        ;*** File m0251.c   , Line 4     ; block
  00000000              _sum:                            ; function: sum
                        ;*** File m0251.c   , Line 5     ; block
                        ;*** File m0251.c   , Line 9     ; expression statement
  00000000 1911                 SUB.W       R1,R1
                        ;*** File m0251.c   , Line 10    ; expression statement
  00000002 1988                 SUB.W       E0,E0
                        ;*** File m0251.c   , Line 10    ; for
  00000004 4004                 BRA         L8:8
  00000006              L7:
                        ;*** File m0251.c   , Line 10    ; block
                        ;*** File m0251.c   , Line 11    ; expression statement
  00000006 0981                 ADD.W       E0,R1
                        ;*** File m0251.c   , Line 10    ; expression statement
  00000008 0B58                 INC.W       #1,E0
  0000000A              L8:
  0000000A 1D08                 CMP.W       R0,E0
  0000000C 4FF8                 BLE         L7:8
                        ;*** File m0251.c   , Line 13    ; return
  0000000E 0D10                 MOV.W       R1,R0
                        ;*** File m0251.c   , Line 14    ; block
  00000010 5470                 RTS
```

【注】

[1] 各セクションのセクション属性（P，C，D，B）
[2] 各セクションの先頭からのオフセット
[3] 各セクションのオフセットアドレスの内容
[4] 機械語に対応するアセンブリコード

図 1.8 **ソースプログラムリストが出力されない場合のオブジェクト情報**

（show＝nosource, object, cpu＝2600a）

(6) 統計情報

統計情報の出力例を**図** 1.9 に示します。

```
******* SECTION SIZE INFORMATION *******

PROGRAM  SECTION(P):                              0x00000012 Byte(s)
CONSTANT SECTION(C):                              0x00000000 Byte(s)
DATA     SECTION(D):                              0x00000000 Byte(s)
BSS      SECTION(B):                              0x00000000 Byte(s)

TOTAL PROGRAM  SECTION: 0x00000012 Byte(s)                              [1]
TOTAL CONSTANT SECTION: 0x00000000 Byte(s)
TOTAL DATA     SECTION: 0x00000000 Byte(s)
TOTAL BSS      SECTION: 0x00000000 Byte(s)

    TOTAL PROGRAM SIZE: 0x00000012 Byte(s)


** ASSEMBLER/LINKAGE EDITOR LIMITS INFORMATION **

NUMBER OF EXTERNAL REFERENCE  SYMBOLS:        0
NUMBER OF EXTERNAL DEFINITION SYMBOLS:        1  [2]
NUMBER OF INTERNAL/EXTERNAL SYMBOLS:          3


********* CPU MODE INFORMATION *********

cpu=2600a   [3]
```

【注】

[1] 各セクションのサイズとその合計
[2] オブジェクトプログラムの外部参照シンボルの数、外部定義シンボルの数、内部ラベルと外部ラベルの合計数
[3] cpuオプション／環境変数で指定されたCPU／動作モード

図 1.9 **統計情報**

■注意

オプション noobject 指定時およびエラーレベル、フェータルレベルのエラーが発生した場合には、統計情報を出力しません。また、オプション code＝asmcode 指定時には、統計情報のセクションサイズ情報（SECTION SIZE INFORMATION）を出力しませんので注意してください。

2. プログラミング

第 2 章　目次

2.1 C コンパイラの限界値 43
2.2 C プログラムの実行方式 45
2.2.1 オブジェクトプログラムの構造 45
2.2.2 データの内部表現 47
2.2.3 アセンブリプログラムとの結合 52
2.3 拡張機能 62
2.3.1 #pragma 拡張子 62
2.3.2 組み込み関数 81
2.3.3 簡易入出力関数のサポート 94
2.4 プログラム作成上の注意事項 95
2.4.1 コーディング上の注意事項 95
2.4.2 プログラム開発上の注意事項 97

2.1 C コンパイラの限界値

C コンパイラがコンパイルできるソースプログラムの限界値を**表** 2.1 に示します。ソースプログラムを作成する場合は、この限界値の範囲内で作成してください。

表 2.1 C **コンパイラの限界値**

No.	分 類	項 目		限界値
1	C コンパイラの起動	一度にコンパイルできるソースプログラムの数		制限なし*2
2		define オプションで指定できるマクロ名の総数		制限なし
3		ファイル名の長さ		128 文字
4	ソースプログラムの行数	1 行の長さ		4096 文字
5		１ファイルあたりのソースプログラムの行数		65535 行
6		コンパイル可能なソースプログラムの行数		制限なし
7	プリプロセッサ	#include 文によるファイルのネストの深さ		30 レベル
8		#define 文によるマクロ名の総数*1		制限なし
9		マクロ定義、マクロ呼び出しで指定できる引数の数		63 個
10		マクロ名の再置き換えの数		32 回
11		#if、#ifdef、#ifndef、#else、#elif 文のネストの深さ		32 レベル
12		#if、#elif 文で指定できる演算子と被演算子の合計数		512 個
13	宣言	関数定義の数		512 個
14		外部結合となる識別子（外部名）の数		12287 個
15		一つの関数内で有効な識別子（内部名）の数		12288 個
16		基本型を修飾するポインタ型、配列型、関数型の合計数		16 個
17		配列の次元数		6 次元
18		配列、構造体のサイズ*3	H8S/2600 ノーマルモード、H8S/2000 ノーマルモード、H8/300H ノーマルモード、H8/300	65535 バイト
			H8/300H アドバンストモード	16777215 バイト
			H8S/2600 アドバンストモード、H8S/2000 アドバンストモード	4294967295 バイト
19	文	複文のネストの深さ		32 レベル
20		繰り返し文（while 文、do 文、for 文）、選択文（if 文、switch 文）の組み合わせによる文のネストの深さ		32 レベル
21		一つの関数内で指定できる goto ラベルの数		511 個
22		switch 文の数		256 個
23		switch 文のネストの深さ		16 レベル
24		case ラベルの数		511 個

No.	分　類	項　　目	限界値
25	文	for 文のネストの深さ	16 レベル
26	式	文字列の長さ	512 文字
27		関数定義、関数呼び出しで指定できる引数の数	63 個
28		一つの式で指定できる演算子と被演算子の合計数	約 500 個
29	標準ライブラリ	open 関数で一度にオープンできるファイルの数	可変*4

【注】　*1 C コンパイラは、あらかじめ 6 個のマクロ名（_ _LINE_ _、_ _FILE_ _、_ _DATE_ _、_ _TIME_ _、_ _STDC_ _、_ _CPU_ _）を登録します。

*2 PC 版はコマンドラインの制約により 127 文字までの入力となります。

*3 アドバンストモードの場合、アドレス空間のビット幅を指定すると、アドレス空間のビット幅に対応するアドレス空間サイズが優先します。

*4 open 関数で一度にオープンできるファイルの数が指定できるようになりました。
詳細は「3.4（4）（a）標準入出力の初期設定ルーチン（_INIT_IOLIB）の作成例」を参照してください。

2.2 C プログラムの実行方式

本節では、C コンパイラが生成するオブジェクトプログラムについて説明します。特に、C プログラムとアセンブリプログラムを結合する場合や、H8S/2600、H8S/2000 および H8/300H、H8/300 を用いたシステムにプログラムを組み込む場合（**「3. システム組み込み」**）に必要になる事項を詳しく説明します。

本節で述べる項目は、以下のとおりです。

2.2.1 オブジェクトプログラムの構造

C プログラム、標準ライブラリが使用するメモリ領域の性質について述べます。

2.2.2 データの内部表現

C プログラムが用いるデータ型のメモリ上での表現について述べます。C プログラムとハードウェア、アセンブリプログラムの間でデータを相互参照するときに必要です。

2.2.3 アセンブリプログラムとの結合

C プログラムで使用する変数名や関数名のうち、他のオブジェクトプログラムとの間で相互に参照できる名前の規則について述べます。また、C プログラムの関数呼び出しでの引数やリターン値の受け渡し方法、レジスタの使用方法に関する規則について述べます。これらは、C プログラムの関数とアセンブリプログラムのルーチン間で相互に呼び出しや参照を行なうときに必要です。

本節では、H8S/2600、H8S/2000、H8/300H または H8/300 のハードウェアの知識を必要としますので、**「ハードウェアマニュアル」**をあわせてお読みください。

2.2.1 オブジェクトプログラムの構造

ここでは、C プログラム、標準ライブラリが使用するメモリ領域の性質について述べます。メモリ領域の性質には、以下の項目があります。

(1) セクション

メモリ領域のうち、C コンパイラが静的に割り付ける領域は、セクションを構成します。セクションにはセクション名とセクション種別があります。セクション名はコンパイラオプション section または拡張言語仕様#pragma section などで変更することができます。

(2) 書き込み操作

プログラム実行時における書き込み操作の可／不可を示します。

(3) 初期値の有無

プログラム実行開始時の初期値の有無です。

（4）境界調整数

データを割り付けるアドレスに関する制約です。

C プログラム、標準ライブラリが使用するメモリ領域の種類とその性質の概要を**表** 2.2 に示します。

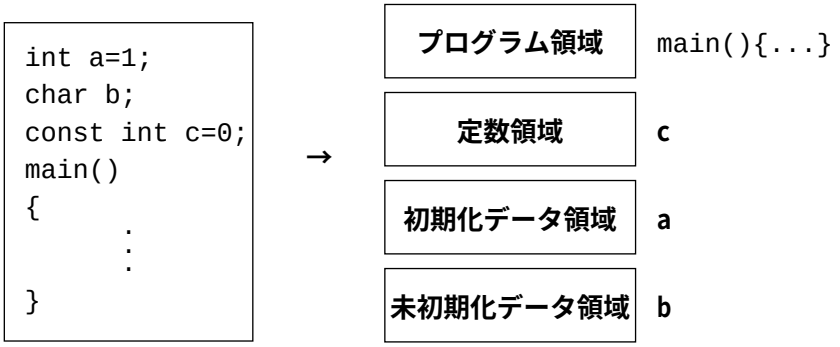
表 2.2　**メモリ領域の種類とその性質の概要**

No.	名　　称	セクション		書き込み操作	初期値の有無	境界調整数	内　　容
		セクション名*	種別				
1	プログラム領域	P	code	不可	有	2byte	機械語を格納する。
2	定数領域	C	data	不可	有	2byte	const 型のデータを格納する。
3	初期化データ領域	D	data	可	有	2byte	初期値のあるデータを格納する。
4	未初期化データ領域	B	data	可	無	2byte	初期値のないデータを格納する。
5	スタック領域	―	―	可	無	―	C プログラム実行に必要な領域。実行時に確保。「3.2.2 動的領域の割り付け」参照。
6	ヒープ領域	―	―	可	無	―	C ライブラリ関数（malloc、realloc、calloc）で使用する領域。「3.2.2 動的領域の割り付け」参照。

【注】　＊　セクション名はコンパイラオプション section または拡張言語仕様#pragma section などで特定の名前を指定しないときに C コンパイラがデフォルトで作成する名前を示します。

■例

CプログラムとCコンパイラの生成するセクションとの対応をプログラム例を用いて示します。



2.2.2 データの内部表現

本節では、C 言語のデータ型と、データの内部表現の対応について述べます。データの内部表現は、以下の項目から成り立っています。

(1) データのサイズ

データの占有する領域のサイズです。

(2) データの境界調整数

データを割り付けるアドレスに関する制約です。任意のアドレスに割り付ける 1 バイト境界調整と、偶数バイトに割り付ける 2 バイト境界調整があります。

(3) データの範囲

スカラ型の値のとり得る範囲を示します。

(4) データの割り付け例

複合型の要素となるデータの割り付け方を示します。

（1）スカラ型

C 言語におけるスカラ型の内部表現を**表** 2.3 に示します。

表 2.3　**スカラ型の内部表現**

No.	データ型	サイズ（byte）	境界調整数（byte）	符号の有無	データの範囲	
					最小値	最大値
1	char	1	1	有	-2^7（-128）	2^7-1（127）
2	signed char	1	1	有	-2^7（-128）	2^7-1（127）
3	unsigned char	1	1	無	0	2^8-1（255）
4	short	2	2	有	-2^{15}（-32768）	2^{15}-1（32767）
5	unsigned short	2	2	無	0	2^{16}-1（65535）
6	int	2	2	有	-2^{15}（-32768）	2^{15}-1（32767）
7	unsigned int	2	2	無	0	2^{16}-1（65535）
8	long	4	2	有	-2^{31}（-2147483648）	2^{31}-1（2147483647）
9	unsigned long	4	2	無	0	2^{32}-1（4294967295）
10	enum　*1	2	2	有	-2^{15}（-32768）	2^{15}-1（32767）
11	float	4	2	有	-∞	+∞
12	double, long double	8	2	有	-∞	+∞
13	**ポインタ（**H8S/2600 **用ノーマルモード、**H8S/2000 **用ノーマルモード、**H8/300H **用ノーマルモード、**H8/300 **用）**	2	2	無	0	2^{16}-1（65535）
14	**ポインタ（**H8/300H **用アドバンストモード）**	4	2	無	0	*2 2^{24}-1（16777215）
15	**ポインタ（**H8S/2600 **用アドバンストモード、**H8S/2000 **用アドバンストモード）**	4	2	無	0	2^{32}-1（4294967295）

【注】　*1 byteenum **オプションを指定した場合、サイズと境界調整数は** 1（byte）**となります。**

*2 **下位** 3 **バイトがアドレスデータで、上位** 1 **バイトは不定値です。**

(2) 複合型

本項では、配列型、構造体型および共用体型の内部表現について説明します。

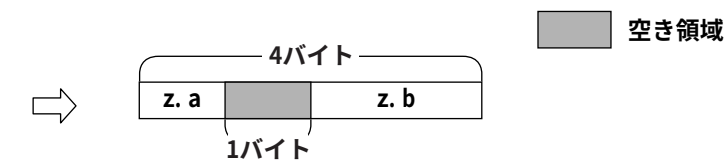
表 2.4 に複合型の内部表現を示します。

表 2.4　**複合型の内部表現**

No.	データ型	境界調整数 (byte)	サイズ (byte)	データの割り付け例		
1	配列型	配列要素の境界調整数	配列要素の数 ×要素サイズ	char a[10];	境界調整数	1byte
					サイズ	10byte
2	構造体型	構造体メンバの境界調整数のうち最大値	メンバのサイズの和 *1	struct { char a,b; };	境界調整数	1byte
					サイズ	2byte
3	共用体型	共用体メンバの境界調整数のうち最大値	メンバのサイズの最大値 *2	union { char a,b; };	境界調整数	1byte
					サイズ	1byte

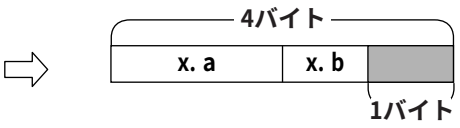
【注】　*1 **構造体型の各メンバを割り付ける時、そのメンバのデータ型の境界調整数に合わせるために直前のメンバとの間に** 1 **バイトの空き領域が生じる場合があります。**

```
struct  {
     char a;
     int b;
  }z;
```



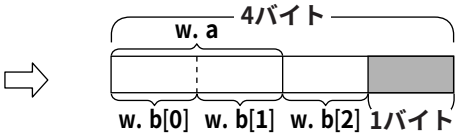
また、構造体が 2 **バイトの境界調整数を持つ場合で、最後のメンバが奇数バイト目で終わっているとき、その次のバイトも含めて構造体型の領域として扱います。**

```
struct  {
     int a;
     char b;
  }x;
```



*2 **共用体が** 2 **バイトの境界調整数を持つ場合で、メンバのサイズの最大値が奇数バイトのとき、その次のバイトも含めて共用体型の領域として扱います。**

```
union  {
     int a;
     char b[3];
  }w;
```



(3) ビットフィールド

ビットフィールドは、構造体の中にビット幅を指定して割り付けるメンバです。
本項では、ビットフィールド特有の割り付け規則について説明します。

(a) ビットフィールドのメンバ

表 2.5 にビットフィールドメンバの仕様を示します。

表 2.5　**ビットフィールドメンバの仕様**

No.	項　　目	仕　　様
1	**ビットフィールドで許される型指定子**	char, unsigned char, short, unsigned short, int, unsigned int
2	**宣言された型に拡張するときの符号の扱い** *1	**符号なし（**unsigned **を指定した型）→ゼロ拡張** *2 **符号あり（**unsigned **を指定しない型）→符号拡張**

【注】 *1 **ビットフィールドのメンバを使用する場合は、ビットフィールドに格納したデータを、宣言した型に拡張して使用します。**

*2 **ゼロ拡張：　拡張するときに上位のビットにゼロを補います。**
符号拡張：　拡張するときにビットフィールドデータの最上位ビットを符号として解釈し、上位のビットに符号ビットを補います。

■注意

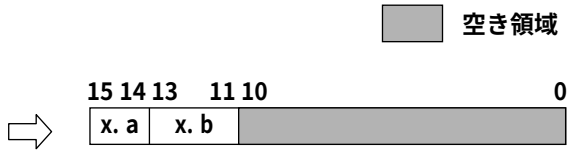
符号付き（signed）で宣言されたサイズが 1 ビットのビットフィールドのデータは、データそのものを符号として解釈します。したがって、表現できる値は 0 と－1 だけになります。0 と 1 を表現する場合には、必ず符号なし（unsigned）で宣言してください。

(b) ビットフィールドの割り付け方

ビットフィールドは、以下の５つの規則に従って割り付けます。

(i) ビットフィールドのメンバは領域内で左（上位ビット側）から順に詰め込みます。
例：

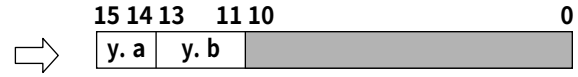
```
struct b1{
        int a:2;
        int b:3;
    }x;
```



(ii) 同じサイズの型指定子が連続している場合は、可能な限り同じ領域に詰め込みます。

例：

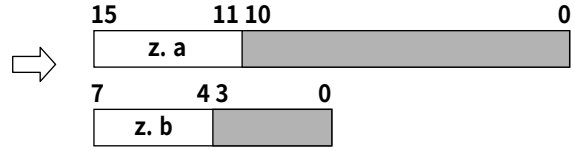
```
struct b1{
        int            a:2;
        unsigned short b:3;
    }y;
```



(iii) 異なるサイズの型指定子で宣言されたメンバは、次の領域に割り付けます。

例：

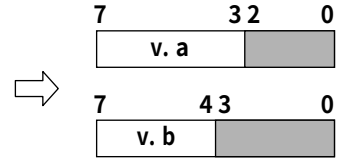
```
struct b1{
        int  a:5;
        char b:4;
    }z;
```



(iv) 同じサイズの型指定子が連続していても、詰め込み先の領域の残りビットが、次のビットフィールドのサイズより小さい場合は、残りの領域は未使用領域となり、次の領域に割り付けます。

例：

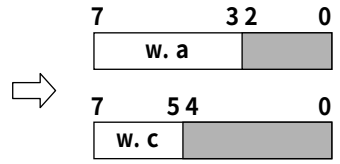
```
struct b2{
      char a:5;
      char b:4;
  }v;
```



(v) 無名のビットフィールドあるいはビット幅0のビットフィールドのメンバを指定すると、次のメンバからは、強制的に次の領域に割り付けます。

例：

```
struct b2{
      char a:5;
      char  :0;
      char c:3;
  }w;
```



2.2.3 アセンブリプログラムとの結合

C 言語はシステムプログラムの記述に適しており、マイコン組み込み用の応用システムのほとんどの処理を C 言語で記述することができます。特に本 C コンパイラでは、アセンブラ埋め込み機能や組み込み関数などをサポートすることにより、全ての機能を C 言語で記述できるようになっています。

しかしながら、ハードウェアのタイミング要求やメモリサイズの制限などのように性能要求が厳しい場合、アセンブリ言語で記述し、C プログラムと結合する必要があります。

ここでは、C プログラムとアセンブリプログラムの結合時に留意すべき以下の内容について述べます。

（1）外部名の相互参照方法
（2）関数呼び出しのインタフェース

（1）外部名の相互参照方法

C プログラムの中で外部名として宣言されたものは、アセンブリプログラムとの間で相互に参照あるいは更新することができます。C コンパイラは、次のものを外部名として扱います。

・大域変数であって、かつ static 記憶クラスでないもの
・extern 記憶クラスで宣言されている変数名
・static 記憶クラスを指定されていない関数名

外部名となる変数名、関数名についての詳細は、**「H シリーズ C 言語マニュアル　6.1　記憶クラス指定子　識別子の結合」**を参照してください。

外部名となる変数名、関数名をアセンブリプログラムで指定する場合は、C プログラム内での名前（31 文字までが有効です）の先頭に下線（ _ ）をつけたものになります。

■例1　アセンブリプログラムの外部名をCプログラムで参照する方法

- アセンブリプログラムでは、「.EXPORT」制御命令を用いてシンボル名（先頭に下線（ _ ）を付与）を外部定義宣言します。
- C プログラムでは、シンボル名（先頭に下線（ _ ）がない）を「extern」宣言します。

アセンブリプログラム（定義する側）

```
    .EXPORT  _a, _b
    .SECTION D,DATA,ALIGN=2
_a: .DATA.W  1
_b: .DATA.W  1
    .END
```

Cプログラム（参照する側）

```
extern int a,b;


f()
{
       a+=b;
}
```

■例2　Cプログラムの外部名をアセンブリプログラムから参照する方法

- C プログラムでは、シンボル名（先頭に下線（ _ ）がない）を外部定義します。
- アセンブリ プログラムでは、「.IMPORT」制御命令を用いてシンボル名（先頭に下線（ _ ）を付与）を外部参照宣言します。

Cプログラム（定義する側）

```
char a,b;
```

アセンブリプログラム（参照する側）

```
  .IMPORT  _a,_b
  .SECTION P,CODE,ALIGN=2
  MOV.B    @_a,R5L
  MOV.B    R5L,@_b
  RTS
  .END
```

（2）関数呼び出しのインタフェース

C プログラムとアセンブリプログラム間で相互に関数呼び出しを行なうときに、アセンブリプログラム側で守るべき次の 4 つの規則について説明します。

(a) スタックポインタに関する規則
(b) スタックフレームの割り付け、解放に関する規則
(c) レジスタに関する規則
(d) 引数、リターン値の設定、参照に関する規則

(a) スタックポインタに関する規則

スタックポインタの指すアドレスよりも下位（0 番地の方向）のスタック領域に、有効なデータを格納してはいけません。スタックポインタより下位アドレスに格納されたデータは、割り込み処理で破壊される可能性があります。

(b) スタックフレームの割り付け、解放に関する規則

関数呼び出しが行なわれた時点（JSR または BSR 命令の実行直後）では、スタックポインタはリターンアドレスの領域を指しています。この領域より上位アドレスのデータの割り付け、設定は呼び出す側の関数の役目です。

関数のリターン時は、リターンアドレスの領域の解放を呼び出される側の関数で行ないます。これは、通常 RTS 命令を用いて行ないます。これより上位アドレスの領域（リターン値アドレスおよび引数領域）は、呼び出した側の関数で解放します。

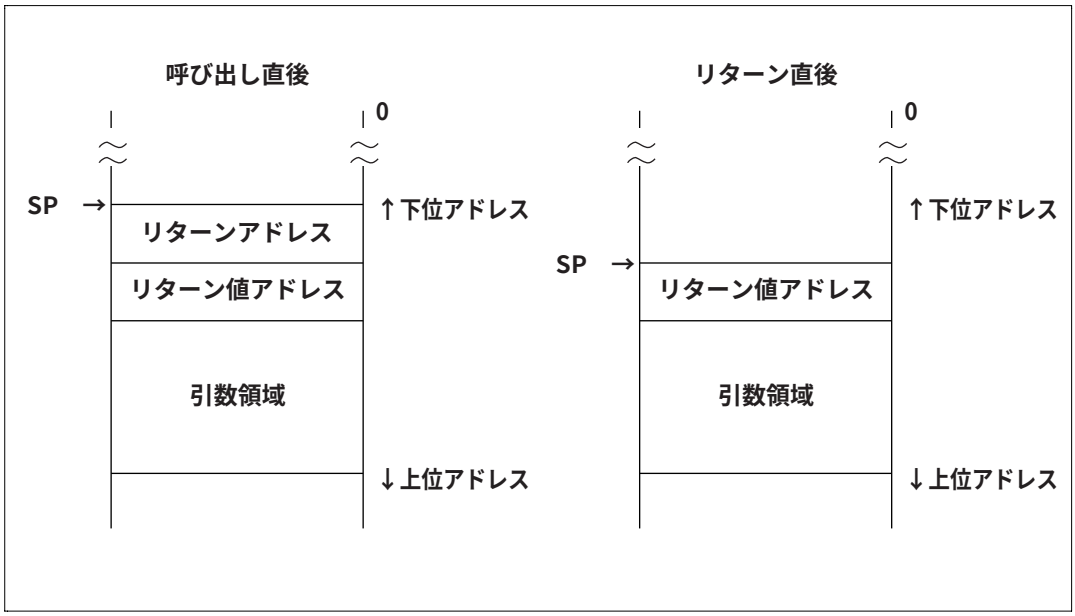


図 2.1　スタックフレームの割り付け、解放に関する規則

(c) レジスタに関する規則

関数呼び出し前後において、値を保証するレジスタと保証しないレジスタがあります。各 CPU 種類におけるレジスタの保証規則を**表** 2.6 に示します。

表 2.6　**関数呼び出し前後のレジスタ保証規則**

No.	項　目	CPU 種類と対象レジスタ		プログラミングにおける留意点
		H8S/2600 用、H8S/2000 用、H8/300H 用	H8/300 用	
1	保証しないレジスタ	ER0、ER1	R0、R1	関数呼び出し時に対象レジスタに有効な値があれば、呼び出し側で値を退避する。呼び出される側の関数では、退避せずに使用可能。
2	保証するレジスタ	ER2～ER6	R2～R6	対象レジスタのうち関数内で使用するレジスタの値を退避し、リターン時に回復する。

以下、レジスタ保証規則について H8S/2600 用アドバンストモードの場合の具体例を示します。

■例 1：**アセンブリプログラムのサブルーチンを** C **プログラムから呼び出す場合**

アセンブリプログラム（呼び出される側）

```
        .EXPORT  _sub
        .SECTION P,CODE,ALIGN=2
_sub:   STM.L    (ER4-ER6),@-SP
        SUB.L    #10,SP
                 :
        ADD.L    #10,SP
        LDM.L    @SP+,(ER4-ER6)
        RTS
        .END
```

} 関数内で使用するレジスタの退避
} 関数本体の処理
（ER0，ER1は保証しないレジスタのため、退避せずに使用可能）
} 退避したレジスタの回復

Cプログラム（呼び出す側）

```
extern void sub();
f()
{
        sub();
}
```

■例2：Cプログラムのサブルーチンをアセンブリプログラムから呼び出す場合

Cプログラム（呼び出される側）

```
void sub()
{
            .
            .
            .
}
```

アセンブリプログラム（呼び出す側）

```
     .IMPORT  _sub
     .SECTION P,CODE,ALIGN=2
            .
            .
     MOV.L    ER1,@(4,SP)
     MOV.L    ER0,ER6
     JSR      @_sub
            .
            .
            .
     .END
```

レジスタER0，ER1に有効な値があれば空き
レジスタまたはスタックに退避
関数「sub」の呼び出し

（d）引数とリターン値の設定、参照に関する規則

以下、引数とリターン値の設定、参照法について説明します。引数とリターン値の規則は、関数の宣言において、個々の引数とリターン値の型が明示的に宣言されているかどうかによって異なります。引数とリターン値の型を明示的に宣言するには、関数の原型宣言を用います。

以下の解説では、まず引数とリターン値に対する一般的な規則について述べたあと、引数の割り付け方とリターン値の設定場所について述べます。

（i）引数とリターン値に対する一般的な規則

引数の渡し方

引数の値を、必ず引数の割り付け領域にコピーしたあとで関数を呼び出します。呼び出した側の関数では、リターン後に引数の割り付け領域を参照することはありませんので、呼び出された側の関数で引数の値を変更しても呼び出した側の処理は直接には影響を受けません。

型変換の規則

引数を渡す場合、またはリターン値を返す場合、自動的に型変換を行なう場合があります。以下、この型変換の規則について説明します。

●リターン値の型変換

リターン値は、その関数の返す型に変換します。

●型の宣言された引数の型変換

原型宣言によって型が宣言されている引数は、宣言された型に変換します。

●型の宣言されていない引数の型変換

原型宣言によって型が宣言されていない引数の型変換は、以下の規則に従って変換します。

・char 型、unsigned char 型の引数は、int 型に変換します。
・float 型の引数は、double 型に変換します。
・上記以外の型は、変換しません。

■例

(1)
```
long f();

long f()
 {    float x;
      return x;
 }
```
リターン値はlong型に型変換します。

(2)
```
void p(int,...);

f()
 {    char c;
      p(1.0, c);
 }
```
cは、対応する引数の型宣言がないので、int型に変換します。

1.0は、対応する引数の型がint型なので、int型に変換します。

■注意

C コンパイラでは、原型宣言によって引数の型を宣言していない場合、正しく引数が渡されるように呼び出される側と呼び出す側で同じ型を指定しないと、動作を保証しません。

```
f(x)
float x;
{
      .
      .
      .
 }

main()
{
   float x;
   f(x);
 }
```

動作を保証しない指定例

```
f(float x)
{
      .
      .
      .
 }

main()
{
   float x;
   f(x);
 }
```

正しい指定例

動作を保証しない指定例では、関数「f」の引数の原型宣言がないため、関数「main」の側で呼び出すときに引数 x を double 型に変換します。

一方、関数「f」の側では引数を float 型として宣言していますので正しく引数を受け渡しすることはできません。

原型宣言によって引数の型を宣言するか、関数「f」の側の引数宣言を double 型にする必要があります。

正しい指定例は、原型宣言によって引数の型を宣言した例です。

(ii) 引数の割り付け領域

引数は、スタック上の引数領域に割り付ける場合と、レジスタに割り付ける場合があります。

オブジェクト種類ごとの引数の割り付け領域を**図** 2.2 に、引数割り付け領域の一般規則を**表** 2.7 にそれぞれ示します。

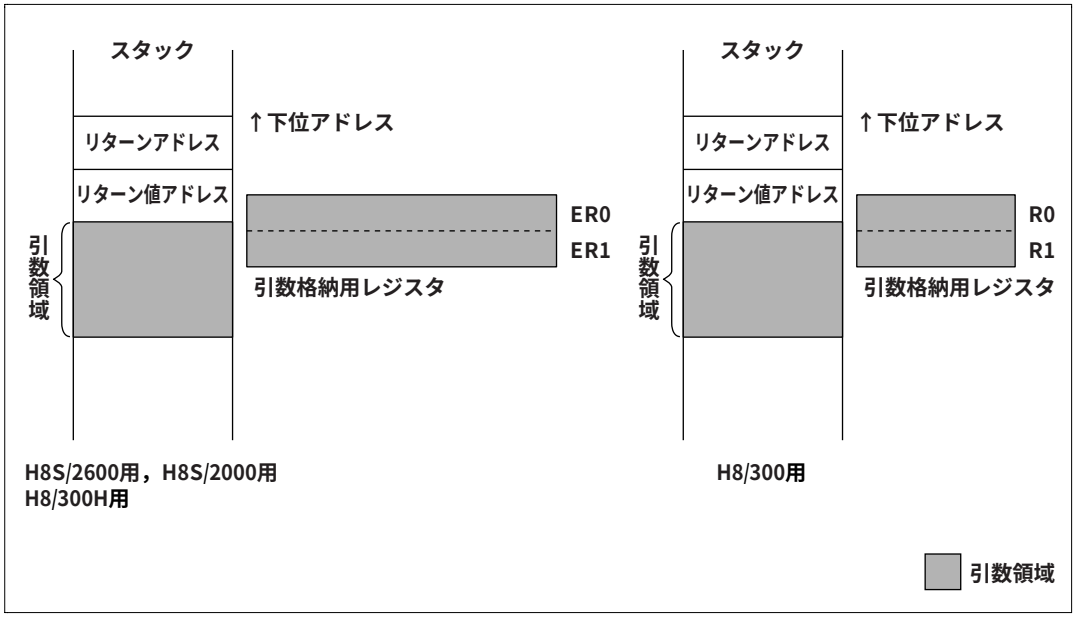


図 2.2　**引数の割り付け領域**

表 2.7　**引数割り付け領域の一般規則**

No.	CPU **種類**	**割り付け規則**		
		レジスタに割り付ける引数		**スタックに割り付ける引数**
		引数格納用レジスタ	**対象の型**	
1	H8S/2600 **用、** H8S/2000 **用、** H8/300H **用**	ER0、ER1	char、unsigned char、short、unsigned short、int、unsigned int、long、unsigned long、float、**ポインタ**	[1] **引数の型がレジスタ渡しの対象の型以外のもの** [2] **原型宣言により可変個の引数をもつ関数として宣言しているもの***
2	H8/300 **用**	R0、R1	char、unsigned char、short、unsigned short、int、unsigned int、**ポインタ**	[3] **引数の数が多いため、レジスタに割り付かなかったもの**

【注】　*　**原型宣言により可変個の引数をもつ関数として宣言している場合、宣言の中で対応する型のない引数およびその直前の引数はスタックに割り付けます。**

例：

```
int f2(int,int,...);
       :
       f2(x,y,z);   →  y,z はスタックに割り付けます。
       :
```

（iii）引数の割り付け

引数格納用レジスタへの割り付け

引数格納用レジスタには、ソースプログラムの宣言順に番号の小さい、LSB側のレジスタから割り付けます。引数格納用レジスタの割り付け例を**図** 2.3 に示します。

```
f(char a,int b)
{
      :
}
```

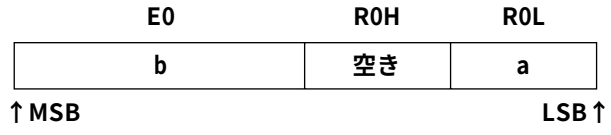


図 2.3　**引数格納用レジスタの割り付け例**（H8S/2600 **用**）

スタック上の引数領域への割り付け

スタック上の引数領域には、ソースプログラム上で指定した順に下位アドレスから割り付けます。

■注意

構造体型、共用体型引数に関する注意

構造体型、共用体型の引数を設定する場合は、その型の本来の境界調整にかかわらず2バイト境界に割り付け、しかもその領域として偶数バイトの領域を使用します。これは、H8S、H8/300シリーズのスタックポインタが2バイト単位で変化するためです。

「付録 B**．引数割り付けの具体例」**に、各CPU／動作モードに対する引数割り付けの具体例がありますので、合わせて参照してください。

（iv）リターン値の設定場所

関数のリターン値の型により、リターン値をレジスタに設定する場合とメモリに設定する場合があります。リターン値の型と設定場所の関係は**表** 2.8 を参照してください。

関数のリターン値をメモリに設定する場合、リターン値はリターン値アドレスの指す領域に設定します。呼び出す側では、引数領域のほかにリターン値設定領域を確保し、そのアドレスをリターン値アドレスの領域に設定してから関数を呼び出します（**図** 2.4 参照）。関数のリターン値がvoid型の場合、リターン値を設定しません。

表 2.8　**リターン値の型と設定場所**

No.	リターン値の型	リターン値の設定場所	
		H8S/2600 用、H8S/2000 用、H8/300H 用	H8/300 用
1	char、unsigned char	レジスタ（R0L）	レジスタ（R0L）
2	short、unsigned short、int、unsigned int	レジスタ（R0）	レジスタ（R0）
3	ポインタ	レジスタ ノーマルモード：（R0） アドバンストモード：（ER0）	レジスタ（R0）
4	long、unsigned long、float	レジスタ（ER0）	リターン値設定領域（メモリ）
5	double、long double、構造体、共用体	リターン値設定領域（メモリ）	リターン値設定領域（メモリ）

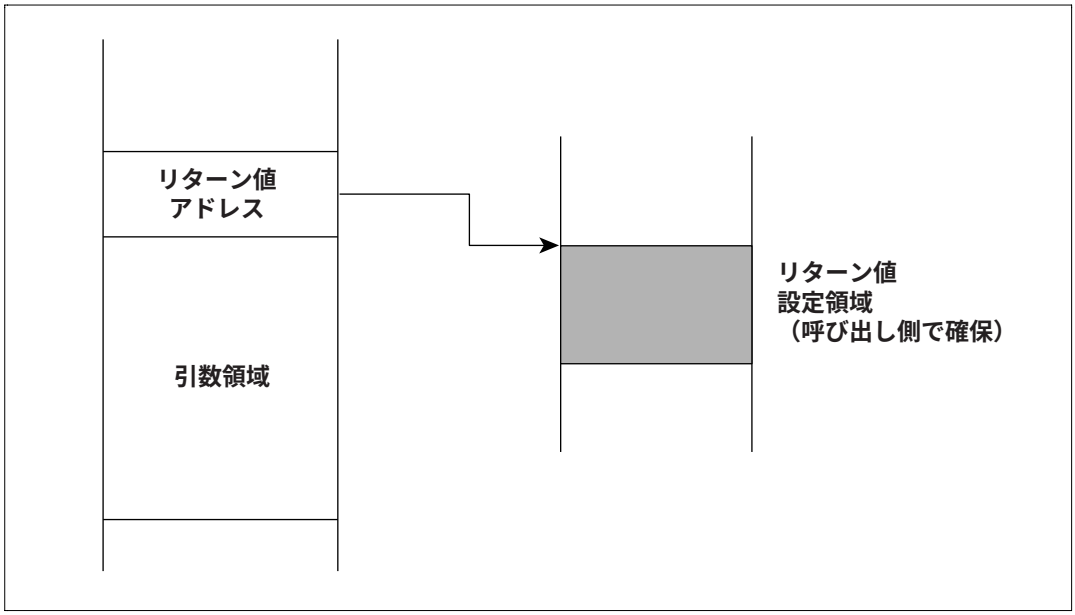


図 2.4　**リターン値をメモリに設定する場合の設定領域**

2.3 拡張機能

C コンパイラの拡張機能として、次の機能をサポートしています。

（1）#pragma 拡張子
（2）組み込み関数

以下、各々の機能について説明します。

2.3.1 #pragma 拡張子

#pragma 拡張子の一覧を**表** 2.9 に示します。

表 2.9 #pragma **拡張子一覧**

No.	#pragma **拡張子**	**機　　能**
1	#pragma abs8、 #pragma abs16	**短絶対アドレス形式を指定**
2	#pragma asm、 #pragma endasm	**C ソースプログラム中にアセンブリ命令列を記述**
3	#pragma indirect	**メモリ間接での関数呼び出しを指定**
4	#pragma inline	**関数のインライン展開を指定**
5	#pragma inline_asm	**C ソースファイル内でアセンブリ言語記述関数をインライン展開**
6	#pragma interrupt	**割り込み関数を C 言語で記述**
7	#pragma section、 #pragma abs8 section、 #pragma abs16 section、 #pragma indirect section	**C ソースプログラム中でセクションを切り替え**
8	#pragma regsave、 #pragma noregsave	**関数の入口／出口でレジスタの退避／回復コード出力指定**
9	#pragma global_register	**グローバル変数にレジスタを割り付け**

(1) 短絶対アドレス機能

#pragma abs8、#pragma abs16 を用いて 8 ビットまたは 16 ビット短絶対アドレス領域に割り付ける変数を短絶対アドレス形式でアクセスするコードを生成することができます。

短絶対アドレス領域の範囲については、**「付録 F. 短絶対アドレスのアクセス範囲」**を参照してください。

abs8、abs16 オプションでもコンパイル単位に短絶対アドレス形式アクセスコードを生成することができますが、本機能ではより細かく変数単位に指定することができます。以下、短絶対アドレス機能の仕様について説明します。

(a) 記述方法

```
#pragma abs8 （＜変数名＞［, ＜変数名＞...］）
#pragma abs16（＜変数名＞［, ＜変数名＞...］）
```

■注意

#pragma abs8、#pragma abs16 宣言は、#pragma abs8、#pragma abs16 で宣言する変数の宣言前に行なってください。

(b) 説明

#pragma abs8 で宣言された変数は、セクション名“ $ABS8＋C セクション名 ”、“ $ABS8＋D セクション名 ”、“ $ABS8＋B セクション名 ”に出力され、8 ビット絶対アドレス（@aa：8）でアクセスするコードを生成します。

#pragma abs16 で宣言された変数は、セクション名“ $ABS16＋C セクション名 ”、“ $ABS16＋D セクション名 ”、“ $ABS16＋B セクション名 ”に出力され、16 ビット絶対アドレス（@aa：16）でアクセスするコードを生成します。

C セクション名、D セクション名、B セクション名の指定方法については、**「2.3.1 (7) セクション切り替え機能」**の#pragma abs8 section および#pragma abs16 section を参照してください。

また、本機能を使用する場合は、次の注意事項に従ってください。

(c) 使用上の注意事項

- #pragma abs8、#pragma abs16 で宣言できる変数は、静的領域へ割り付ける変数のみです。
- #pragma abs8、#pragma abs16 文 1 行に宣言できる変数の数は 63 個までです。
- #pragma abs8、#pragma abs16 で宣言した変数は、セクション切り替え機能を使用しない場合、セクション名“ $ABS8C ”、“ $ABS8D ”、“ $ABS8B ”、“ $ABS16C ”、“ $ABS16D ”、または“ $ABS16B ”へ出力されます。リンク時には、当該セクションを必ず 8 ビット／16 ビット絶対アドレス領域に割り付けてください。

(d) 短絶対アドレス機能の使用例

■例

```
#pragma abs8(c)
#pragma abs16(i)
char c;
int i;
long l;
f( ){
    c=10;
    i=100;
    l=1000;
}
```

上記、C ソースプログラムを cpu＝2600a でコンパイルしたときのオブジェクトプログラムを以下に示します。

```
        .CPU     2600A
        .EXPORT  _c
        .EXPORT  _i
        .EXPORT  _l
        .EXPORT  _f
        .SECTION P,CODE,ALIGN=2
_f:
        MOV.B    #10:8,ROL
        MOV.B    ROL,@_c:8          ;c を 8 ビット短絶対アドレスでアクセスします。
        MOV.W    #100:16,R0
        MOV.W    R0,@_i:16          ;i を 16 ビット短絶対アドレスでアクセスします。
        MOV.L    #1000:32,ER0
        MOV.L    ER0,@_l:32         ;l を 32 ビット短絶対アドレスでアクセスします。
        RTS
        .SECTION $ABS8B,DATA,ALIGN=2
_c:
        .RES.B   1                  ;c は,セクション名$ABS8B に割り付きます。
        .SECTION $ABS16B,DATA,ALIGN=2
_i:                                 ;i は,セクション名$ABS16B に割り付きます。
        .RES.W   1
        .SECTION B,DATA,ALIGN=2
_l:                                 ;l は,セクション名 B に割り付きます。
        .RES.L   1
        .END
```

(2) アセンブラ埋め込み機能

#pragma asm、#pragma endasm を用いて、C プログラム中にアセンブラ命令列を埋め込むことができます。以下、アセンブラ埋め込み機能の仕様について説明します。

(a) 記述方法

```
#pragma asm
        ＜アセンブラ命令列＞
#pragma endasm
```

(b) 説明

#pragma asm～#pragma endasm で囲まれた範囲にアセンブラ命令列を記述することができます。

C コンパイラは#pragma asm～#pragma endasm で囲まれた命令列を C コンパイラが生成するオブジェクトコードの中に展開します。本機能を使用する場合は、次の注意事項に従ってください。

(c) 使用上の注意事項

・コンパイル時に code＝asmcode オプションを用いてアセンブリプログラムの出力を指定してください。指定がない場合は#pragma asm、#pragma endasm を含むアセンブラ命令列を無視します。

・C コンパイラはアセンブラ命令列の文法や、C コンパイラ生成コードへの影響についてはチェックしません。また、コンパイル時に optimize＝1 オプションや speed オプションを指定した場合、アセンブラ命令列の展開内容や展開位置が実際の指定と一致しない場合があります。アセンブラ埋め込み機能を使用する場合は、出力コードおよびプログラムの動作を十分確認してください。

・#pragma asm～#pragma endasm をネストして指定することはできません。ネストがある場合、エラーを出力します。

・選択文、繰り返し文で#pragma asm～#pragma endasm を指定する場合、#pragma asm、#pragma endasm を含むアセンブラ命令列を複文 { } で囲む必要があります。複文で囲まれていない場合、結果は保証しません。

■例

```
while(a==0)
{                    …… 必ず指定してください。
#pragma asm
        ＜アセンブラ命令列＞
#pragma endasm
}                    …… 必ず指定してください。
```

(d) アセンブラ埋め込みの使用例

■例

```
void main()
#pragma asm
   MOV.L #H'FFFFFFFE,SP ;main関数の始まる前にｽﾀｯｸのｱﾄﾞﾚｽを設定します。
#pragma endasm
{
    :
 }
```

(3) メモリ間接の関数呼び出し

#pragma indirect を用いて、関数をメモリ間接（@@aa：8）で呼び出すことができます。

indirect オプションでもコンパイル単位に関数をメモリ間接で呼び出すことができますが、本機能ではより細かく関数単位に指定することができます。

以下、メモリ間接での関数呼び出し仕様について説明します。

(a) 記述方法

`#pragma indirect`（関数名［, 関数名...］）

■注意

#pragma indirect 宣言は、#pragma indirect で宣言する関数の宣言、定義より前に行なってください。

(b) 説明

#pragma indirect を用いて、メモリ間接呼び出しとなる関数を宣言します。

#pragma indirect を用いて、宣言された関数は、JSR　@@“ $＋関数名 ”：8 の形で呼び出します。また、メモリ間接呼び出し宣言された関数の本体が定義されている場合は、“ $＋関数名 ”のラベルと関数のアドレスが、セクション名“ $INDIRECT＋セクション名 ”にメモリ間接呼び出しのためのアドレステーブルとして確保されます。

セクション名の指定方法については、**「2.3.1 (7) セクション切り替え機能」**の#pragma indirect section を参照してください。

本機能を使用する場合は、次の注意事項に従ってください。

(c) 使用上の注意事項

- #pragma indirect 文 1 行に宣言できる関数の数は 63 個までです。
- #pragma indirect で宣言できる関数の数は、CPU／動作モードが 2600a、2000a、300ha のとき 64 個、また 2600n、2000n、300hn、300 のとき 128 個までです。
- #pragma indirect で宣言した関数のアドレステーブルはセクション切り替え機能を使用しない場合、“ $INDIRECT ”へ出力されます。

・ リンク時にはアドレステーブルのセクションを 0x0000～0x00FF 番地に割り付けてください。

・ #include <indirect.h>を宣言することにより、実行時ルーチンの呼び出しコードをメモリ間接呼び出しにすることができます。
メモリ間接呼び出しを行う実行時ルーチンを選択したい場合には、indirect.h の中で必要な#pragma indirect 文以外をコメントにしてください。

■例

```
#include <indirect.h>
void main(void)
{
        :
}
```

（d）メモリ間接の関数呼び出しの使用例

■例

```
#pragma indirect (f)          /* fをメモリ間接呼び出し宣言します。*/
#define A *((unsigned char *)0xffffff00)
unsigned char a;
unsigned char f(void)
{
    sub( );
    return(A&1);
}
sub( )
{
    a=f( );                   /* 関数呼び出し。*/
}
```

上記、C ソースプログラムを cpu＝2600a でコンパイルしたときのオブジェクトプログラムを以下に示します。

```
        .CPU        2600A
        .EXPORT     _a
        .EXPORT     _f
        .EXPORT     $f                  ;アドレステーブルのラベルを外部定義宣言。
        .EXPORT     _sub
        .SECTION    P,CODE,ALIGN=2
_f:
        BSR         _sub:8
        MOV.B       @-256:8,ROL
        AND.B       #1:8,ROL
        RTS
_sub:
        JSR         @@$f:8              ;メモリ間接で関数呼び出し。
        MOV.B       ROL,@_a:32
        RTS
        .SECTION    $INDIRECT,DATA,ALIGN=2
$f:                                     ;アドレステーブルのラベル。
        .DATA.L     _f                  ;アドレステーブルの定義。
        .SECTION    B,DATA,ALIGN=2
_a:
        .RES.B      1
        .END
```

（4）関数のインライン展開

#pragma inline を用いて、関数をインライン展開することができます。

speed＝inline オプションでも関数のインライン展開ができますが、本機能では最適化指定の有無にかかわらず、また関数単位で指定することができます。

以下、関数のインライン展開仕様について説明します。

（a）記述方法

```
#pragma inline（関数名［, 関数名...］）
```

（b）説明

#pragma inline を用いて、インライン展開となる関数を宣言します。

#pragma inline を用いて宣言した関数を呼び出すと JSR、BSR 命令で関数を呼び出すコードは出力されず、呼び出した場所へ関数のコードが直接展開されます。

本機能を使用する場合は、次の注意事項に従ってください。

（c）使用上の注意事項

（i）#pragma inline 文 1 行に宣言できる関数の数は 63 個までです。

（ii）#pragma inline で宣言された関数が、次のいずれかの条件を満たす時、インライン展開されません。

- 可変引数を持つ。
- 引数のアドレスを参照している。
- 実引数と仮引数の型が不一致である。
- switch 文が記述されている。
- インライン展開の制限サイズを超えている。

（iii）#pragma inline で指定した関数に対しても外部定義を生成します。各ソースプログラムファイル中にインライン関数の実体の記述がある場合は、必ず関数の宣言に static を指定してください。static を指定した場合は、外部定義を生成しません。

(d) 関数のインライン展開の使用例

■例

```
#pragma inline (f)        /* 関数 f をインライン展開として宣言します。*/
int a,b,c;
int f(int x,int y)
{
    return x+y;
}
sub( )
{
    a=f(b,c);             /* 直接 a=b+c のコードに展開されます。*/
}
```

上記、C ソースプログラムを cpu＝2600a でコンパイルしたときのオブジェクトプログラムを以下に示します。

```
      .CPU        2600A
      .EXPORT     _a
      .EXPORT     _b
      .EXPORT     _c
      .EXPORT     _f
      .EXPORT     _sub
      .SECTION    P,CODE,ALIGN=2
_f:
      MOV.W       R0,R1
      ADD.W       E0,R1
      MOV.W       R1,R0
      RTS
_sub:
      MOV.W       @_c:32,E0         ;
      MOV.W       @_b:32,R0         ;
      ADD.W       E0,R0             ; 直接a=b+cのコードに展開します。
      MOV.W       R0,R1             ;
      MOV.W       R1,@_a:32         ;
      RTS
      .SECTION    B,DATA,ALIGN=2
_a:
      .RES.W      1
_b:
      .RES.W      1
_c:
      .RES.W      1
      .END
```

(5) アセンブラ埋め込みインライン展開

C ソースファイル内でアセンブリ言語で記述した関数をインライン展開します。

(a) 記述方法

```
#pragma inline_asm (<関数名>[,<関数名>...])
```

(b) 説明

アセンブラ埋め込みインライン関数のパラメタは、通常の関数呼び出しと同様にレジスタ、あるいはスタックに設定されますので、inline_asm 関数から参照することができます。アセンブラ埋め込みインライン関数のリターン値は(E)R0 に設定してください。

(c) 使用上の注意事項

- #pragma inline_asm は、関数本体の定義の前に指定してください。
- #pragma inline_asm で指定した関数に対しても外部定義を生成します。各ソースプログラムファイル中にインライン関数の実体の記述がある場合は、必ず関数の宣言に static を指定してください。static を指定した場合は、外部定義を生成しません。
- アセンブラ埋め込みインライン関数内でラベルを使用する場合、必ずローカルラベルを使用してください。ローカルラベルの詳細は、**「H8S, H8/300 シリーズクロスアセンブラユーザーズマニュアル」**を参照してください。
- アセンブラ埋め込みインライン関数内で(E)R2 から(E)R6 のレジスタを使用する場合は、アセンブラ埋め込みインライン関数の先頭と最後でこれらレジスタの退避・回復が必要です。
- アセンブラ埋め込みインライン関数の最後に RTS を記述しないでください。
- 本機能を使用する際は、オブジェクト形式指定オプション code = asmcode を用いてコンパイルしてください。
- 本機能を使用した場合、C コンパイラ出力のアセンブリプログラムに対してクロスアセンブラで“ 402 ILLEGAL VALUE IN OPERAND ”のエラーが出ることがあります。これはアセンブラ埋め込み箇所を含んだ分岐幅を 16 ビットディスプレースメントで出力しているため、実際の分岐幅がその範囲を超えると出力されます。分岐幅が届くように JMP 命令を使用して、C コンパイラ出力のアセンブリプログラムを修正してください。

■例

```
（修正前）                          （修正後）
 :                                   :
 BEQ L1        →                     BNE Ld
 :                                   JMP L1
                                   Ld:
```

（d）使用例

・ソースプログラム

```
#pragma inline_asm(shlu)
extern unsigned int x;
static unsigned int shlu(unsigned int a)
{
        SHLL.W  R0
        BCC     ?L1
        SUB.W   R0,R0
   ?L1:
}
void main(void)
{
  x = shlu(x);
}
```

・出力結果

```
          .CPU        2600A
          .IMPORT     _x
          .EXPORT     _main
          .SECTION    P,CODE,ALIGN=2
_main:
          MOV.W   @_x:32,R0
          SHLL.W  R0
          BCC     ?L1
          SUB.W   R0,R0
   ?L1:
          MOV.W   R0,@_x:32
          RTS
          .END
```

（6）割り込み関数の作成機能

本Cコンパイラでは、#pragma interruptを用いて、外部（ハードウェア）割り込み関数をCプログラムで記述することができます。以下、割り込み関数の作成方法を説明します。

（a）記述方法

#pragma interrupt（関数名［（割り込み仕様）］［, 関数名［（割り込み仕様）］...］）

割り込み仕様の一覧を**表** 2.10 に示します。

表 2.10　**割り込み仕様の一覧**

No.	項目	形式	オプション	指定内容
1	スタック切り換え指定	sp＝	＜変数＞ | &＜変数＞ | ＜定数＞ | ＜変数＞＋＜定数＞ | &＜変数＞＋＜定数＞	新しいスタックのアドレスを変数または定数で指定 ＜変数＞　：変数（ポインタ型） &＜変数＞　：変数（オブジェクト型）のアドレス ＜定数＞　：定数値
2	トラップ命令リターン指定	tn＝	＜定数＞	終了を TRAPA 命令で指定 ＜定数＞　：定数値（トラップベクタ番号）
3	割り込み関数終了指定	sy＝	＜関数名＞ | ＜定数＞ | $＜関数名＞	終了を割り込み関数へのジャンプ命令で指定 ＜関数名＞　：割り込み関数名 ＜定数＞　：絶対アドレス $＜関数名＞：下線（ _ ）を付加しない割り込み関数名

（b）説明

#pragma interrupt を用いて割り込み関数となる関数を宣言します。

#pragma interrupt を用いて宣言した関数は、関数の処理の前後で使用しているレジスタを保証（関数入口／出口で関数内で使用する全レジスタを退避・回復）します。また、通常 RTE 命令でリターンしますが、トラップ命令リターン指定（tn＝）をした場合は TRAPA 命令でリターンします。また、割り込み関数終了指定（sy＝）をした場合は JMP 命令で指定されたアドレスへジャンプします。

割り込み関数終了指定の関数名には“ 関数名 ”のみの指定以外に、“ $＋関数名 ”の指定が可能です。“ $＋関数名 ”指定の時は、外部名となる関数名の先頭に下線（ _ ）が付加されません。

割り込み仕様を指定しない場合は単純な割り込み関数として処理します。

また、トラップ命令リターン指定と割り込み関数終了指定は重複して指定できませんが、スタック切り換え指定とトラップ命令リターン指定、およびスタック切り換え指定と割り込み関数終了指定は重複して指定できます。

■例

```
extern char STK[100];
#pragma interrupt ( f(sp=STK+100, tn=2) )
                          [1]       [2]
```

■説明

［1］スタック切り換え指定　STK＋100 を割り込み関数「f」で使用するスタックポインタとして設定します。

［2］トラップ命令リターン指定　割り込み関数終了時に TRAPA #2 でトラップ例外処理を開始します。トラップ例外処理開始時の SP は下図のようになっています。トラップルーチンの側で RTE 命令を使用して PC、CCR（コンディションコードレジスタ）（H8S シリーズは EXR（エクステンドレジスタ）も）を回復し、割り込み関数から復帰してください。

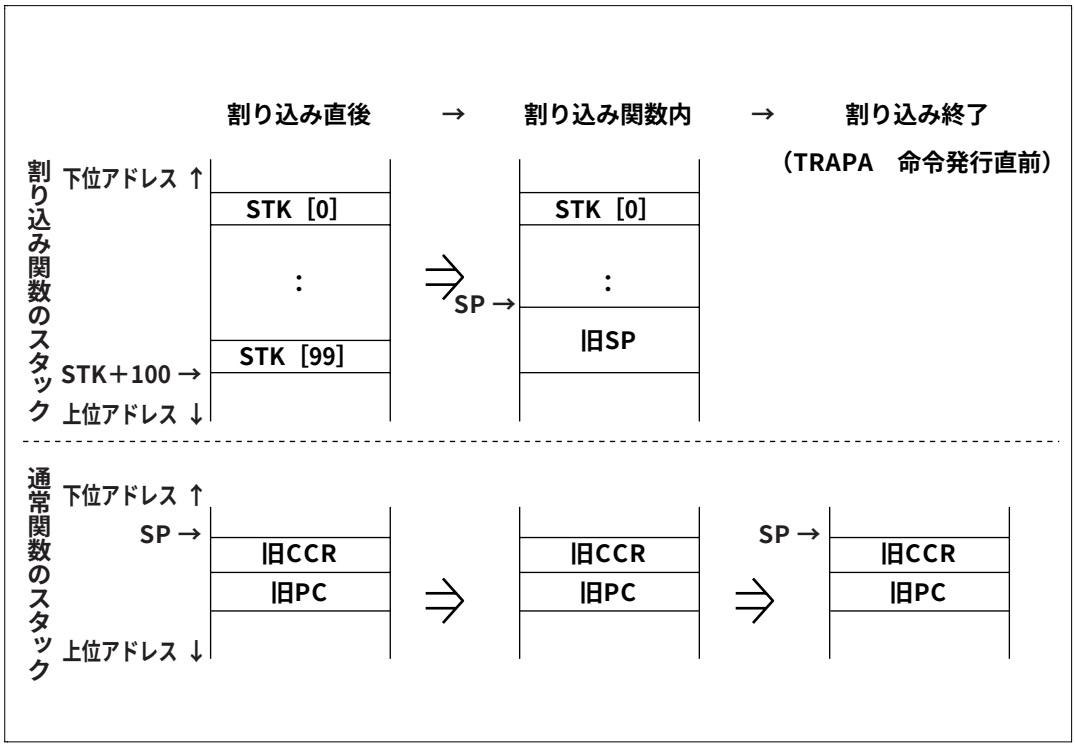


図 2.5　割り込み関数によるスタック使用例

■例

```
#pragma interrupt (f1(sy=$f2))
char x;
void f1(void)
{
    x=0;
}
```

上記、C ソースプログラムを cpu＝2600a でコンパイルしたときのオブジェクトプログラムを以下に示します。

```
     .CPU       2600A
     .IMPORT    f2                ;下線(_)が付加されない
     .EXPORT    _f1
     .EXPORT    _x
     .SECTION   P,CODE,ALIGN=2
_f1:
     PUSH.W     R0
     SUB.B      R0L,R0L
     MOV.B      R0L,@_x:32
     POP.W      R0
     JMP        @f2:24            ;JMP @f2 でリターン
     .SECTION   B,DATA,ALIGN=2
_x:
     .RES.B     1
     .END
```

本機能を使用する場合は、次の注意事項に従ってください。

(c) 使用上の注意事項

・ 割り込み関数の定義に対して指定できる記憶クラス指定子は extern だけで、static を指定しても extern として処理します。
また、関数の返すデータ型は void のみです。return 文のリターン値を指定することはできません。指定があった場合はエラーを出力します。

■例 1

```
#pragma interrupt(f1(sp=100),f2)
void f1( ){…}  ……(a)
int f2( ){…}   ……(b)
```

説明：

（a）は正しい宣言になります。

（b）は関数の返すデータ型は void ではないので誤った宣言です。エラーを出力します。

・ 割り込み関数として宣言した関数をプログラムの中で呼び出すことはできません。呼び出しがあった場合はエラーを出力します。ただし、割り込み関数として定義した関数を、割り込み関数の宣言のないプログラム内で呼び出した場合は、エラーは出力しません。この場合実行時の動作は保証しません。

■例 2：割り込み関数宣言のある場合

```
#pragma interrupt(f1)
void f1(void){…}
int f2( ){ f1( );}  ……(a)
```

説明：

関数「f1」は割り込み関数として宣言しているのでプログラム中で呼び出すことはできません。（a）に対してエラーを出力します。

■例 3：割り込み関数宣言がない場合

```
int f2(){ f1();}   ……(b)
```

説明：

関数「f1」は割り込み関数としての宣言がないので extern int f1（）；を仮定してオブジェクトを生成します。関数「f1」が別コンパイル単位で割り込み関数として宣言された場合、実行時の動作は保証しません。

・ 割り込み関数として宣言した関数に対して、明示的な関数呼び出しによる参照を除いて関数の参照をすることができます。

■例 4：

```
#pragma interrupt f
void f(void)
{
        :
}
void (*VTBL)(void)={f};→ 正常
```

・ #pragma interrupt 文 1 行に宣言できる関数の数は 63 個までです。
・ 割り込み関数にスタック切り替え指定を指定した場合、コンパイルリストのシンボル割り付け情報の Linkage Area Size には、旧 SP と SP 計算のための ER0（H8/300 時は R0）の退避領域のサイズも含まれます。

（7）セクション切り替え機能

#pragma section を用いて、C プログラムの中でコンパイラの出力するセクション名を切り替えることができます。関数単位、変数単位で割り付けるアドレスを指定するためには、従来はファイル分割をしなければなりませんでした。本機能を用いることにより、関数単位、変数単位のファイル分割が不要になります。以下、セクション切り替え機能の仕様について説明します。

(a) 記述方法

```
#pragma section［<名前>｜<数値>］
#pragma abs8 section［<名前>｜<数値>］
#pragma abs16 section［<名前>｜<数値>］
#pragma indirect section［<名前>｜<数値>］
```

(b) 説明

“ #pragma section 名前 ” または “ #pragma section 数値 ” を用いて、セクション名を指定します。ソースプログラム中の宣言位置以降のセクション名が “ P セクション名＋名前（数値） ”、“ C セクション名＋名前（数値） ”、“ D セクション名＋名前（数値） ”、“ B セクション名＋名前（数値） ”、になります。<名前>や<数値>を省略すると、以降はデフォルトのセクション名になります。

“ #pragma abs8 section 名前 ” または “ #pragma abs8 section 数値 ” を用いて、8 ビット絶対アドレス領域のセクション名を指定します。宣言以降の 8 ビット絶対アドレス領域のセクション名が、“ $ABS8C＋名前（数値） ”、“ $ABS8D＋名前（数値） ”、“ $ABS8B＋名前（数値） ” になります。

<名前>や<数値>を省略すると、以降 “ $ABS8C ”、“ $ABS8D ”、“ $ABS8B ” となります。

“ #pragma abs16 section 名前 ”または“ #pragma abs16 section 数値 ”を用いて、16 ビット絶対アドレス領域のセクション名を指定します。宣言以降の 16 ビット絶対アドレス領域のセクション名が、16 ビット絶対アドレスの場合は、“ $ABS16C＋名前（数値）”、、“ $ABS16D＋名前（数値）”、“ $ABS16B＋名前（数値）”になります。＜名前＞や＜数値＞を省略すると、以降“ $ABS16C ”、“ $ABS16D ”、“ $ABS16B ”になります。

“ #pragma indirect section 名前 ”または“ #pragma indirect section 数値 ”を用いて、メモリ間接呼び出し関数のベクタテーブル出力セクション名を指定します。宣言以降のベクタテーブルのセクション名が“ $INDIRECT＋名前（数値）”となります。＜名前＞や＜数値＞を省略すると、以降“ $INDIRECT ”となります。

(c) 使用上の注意事項

・#pragma section、#pragma abs8 section、#pragma abs16 section、#pragma indirect section は関数定義の外で宣言しなければなりません。

・1 ファイルで宣言できるセクション名はそれぞれ最大 64 個です。

(d) セクション切り替え機能の使用例

■例

```
#pragma section abc
int a;              /* a は,セクション Babc に割り付きます。*/
const int c=1;      /* c は,セクション Cabc に割り付きます。*/
f(){                /* f は,セクション Pabc に割り付きます。*/
    a=c;
}
#pragma section
int b;              /* b は,セクション B に割り付きます。*/
g(){                /* g は,セクション P に割り付きます。*/
    b=c;
}
```

上記、C ソースプログラムを cpu＝2600a でコンパイルしたときのオブジェクトプログラム例を以下に示します。

```
                .CPU      2600A
                .EXPORT   _a
                .EXPORT   _c
                .EXPORT   _f
                .EXPORT   _b
                .EXPORT   _g
                .SECTION  Pabc,CODE,ALIGN=2
        _f:
                MOV.W     @_c:32,R0
                MOV.W     R0,@_a:32
                RTS
                .SECTION  P,CODE,ALIGN=2
        _g:
                MOV.W     @_c:32,R0
                MOV.W     RO,@_b:32
                RTS
                .SECTION  Cabc,DATA,ALIGN=2
        _c:
                .DATA.W    H'0001
                .SECTION  Babc,DATA,ALIGN=2
        _a:
                .RES.W    1
                .SECTION  B,DATA,ALIGN=2
        _b:
                .RES.W    1
                .END
```

上記例において、コンパイルオプション section＝p＝PROG が指定された場合、f はセクション PROGabc、g はセクション PROG にそれぞれ割り付きます。

(8) レジスタ退避／回復コード制御機能

#pragma regsave を用いて、関数の入口／出口で関数内で使用しないレジスタを退避／回復するコードを生成することができます。

また、#pragma noregsave を用いて、関数内で使用するレジスタを退避／回復するコードの生成を抑止することができます。

以下、レジスタ退避／回復コード制御機能の仕様について説明します。

(a) 記述方法

```
#pragma regsave（＜関数名＞［, ＜関数名＞...］）
#pragma noregsave（＜関数名＞［, ＜関数名＞...］）
```

■注意

#pragma regsave/noregsave 宣言は、#pragma regsave/noregsave で宣言する関数の定義、宣言前に行なってください。

(b) 説明

#pragma regsave で宣言された関数は、関数内で ER0、ER1（H8/300 では R0、R1）以外のレジスタの使用／未使用にかかわらず最適化有りの時 ER2～ER6（H8/300 では R2～R6）、最適化なしの時 ER2～ER5（H8/300 では R2～R5）を全て関数の入口で退避し、出口で回復するコードを生成します。

#pragma noregsave で宣言された関数は、関数内でレジスタの使用／未使用に関わらず、レジスタの退避／回復コードを生成しません。

■例

```
#pragma regsave (f)          /*レジスタ退避／回復コード生成を宣言*/
f(){}
```

上記、C ソースプログラムを cpu＝2600a でコンパイルしたときのオブジェクトプログラムを次に示します。

```
        .CPU        2600A
        .EXPORT     _f
        .SECTION    P,CODE,ALIGN=2
_f:
        STM.L       (ER2-ER3),@-SP   ┐ レジスタ退避コード
        STM.L       (ER4-ER6),@-SP   ┘
        LDM.L       @SP+,(ER4-ER6)   ┐ レジスタ回復コード
        LDM.L       @SP+,(ER2-ER3)   ┘
        RTS
        .END
```

本機能を使用する場合は、次の注意事項に従ってください。

(c) 使用上の注意事項

・#pragma regsave／noregsave 文 1 行に宣言できる関数の数は 63 個までです。

(9) グローバル変数のレジスタ割り付け

グローバル変数に、レジスタを割り付けます。

(a) 記述方法

```
#pragma global_register (<変数名>=<レジスタ名>[,<変数名>=<レジスタ名>...])
```

(b) 説明

<変数名>で指定したグローバル変数に、<レジスタ名>で指定したレジスタを割り付けます。

(c) 使用上の注意事項

- グローバル変数で、単純型またはポインタ型の変数に使用できます。double 型の変数は指定できません。また、cpu=300 のときは、long 型、float 型の変数も指定できません。
- 指定可能なレジスタは、(E)R4、(E)R5 です。
- 初期値の設定はできません。また、アドレスの参照もできません。
- 指定された変数の、（ファイル内にレジスタ指定のない）リンク先からの参照は保証されません。
- 割り込み関数内での設定・参照は保証されません。
- 変数、レジスタの重複指定、#pragma abs8、#pragma abs16 との二重指定はできません。

(d) 使用例

```
#pragma global_register(x=R4,y=R5L)

int      x;
char     y;

void func1(void)
{
   x++;
}

void func2(void)
{
   y=0;
}

void func(int  a)
{
   x = a;
   func1();
   func2();
}
```

2.3.2 組み込み関数

システム制御命令等、以下に示すC言語で記述できない機能を組み込み関数として提供します。

（1）コンディションコードレジスタの設定・参照
（2）エクステンドレジスタの設定・参照
（3）特殊命令（TRAPA、SLEEP、MOVFPE、MOVTPE、EEPMOV、MAC、NOP）
（4）ローテート演算
（5）コンディションコード反映演算
（6）10進演算

組み込み関数の一覧を**表** 2.11 に示します。

表 2.11 **組み込み関数の一覧**

No.	項　目	仕　様	機　能
1	コンディション	void set_imask_ccr(unsigned char)	割り込みマスクの設定
2	コードレジスタ	unsigned char get_imask_ccr(void)	割り込みマスクの参照
3		void set_ccr(unsigned char)	コンディションコードレジスタの設定
4		unsigned char get_ccr(void)	コンディションコードレジスタの参照
5		void and_ccr(unsigned char)	コンディションコードレジスタの論理積
6		void or_ccr(unsigned char)	コンディションコードレジスタの論理和
7		void xor_ccr(unsigned char)	コンディションコードレジスタの排他的論理和
8	エクステンド	void set_imask_exr(unsigned char)	割り込みマスクの設定
9	レジスタ	unsigned char get_imask_exr(void)	割り込みマスクの参照
10		void set_exr(unsigned char)	コンディションコードレジスタの設定
11		unsigned char get_exr(void)	コンディションコードレジスタの参照
12		void and_exr(unsigned char)	コンディションコードレジスタの論理積
13		void or_exr(unsigned char)	コンディションコードレジスタの論理和
14		void xor_exr(unsigned char)	コンディションコードレジスタの排他的論理和

No.	項　目	仕　様	機　能
15	特殊命令	void trapa(unsigned int)	TRAPA 命令
16		void sleep(void)	SLEEP 命令
17		void movfpe(char*,char)	MOVFPE 命令
18		void movtpe(char,char*)	MOVTPE 命令
19		void tas(char*)	TAS 命令
20		void eepmov(char*,char*,unsigned char) void eepmov(char*,char*,unsigned int)	EEPMOV 命令
21		long mac(long,int*,int*,unsigend long) long macl(long,int*,int*,unsigned long,unsigned long)	MAC 命令
22		void nop(void)	NOP 命令
23	ローテート演算	char rotlc(int,char)	1 バイト左ローテート
24		int rotlw(int,int)	2 バイト左ローテート
25		long rotll(int,long)	4 バイト左ローテート
26		char rotrc(int,char)	1 バイト右ローテート
27		int rotrw(int,int)	2 バイト右ローテート
28		long rotrl(int,long)	4 バイト右ローテート
29	コンディションコード反映演算	int ovfaddc(char,char,char*)	1 バイト加算 ＋コンディションコード反映
30		int ovfaddw(int,int,int*)	2 バイト加算 ＋コンディションコード反映
31		int ovfaddl(long,long,long*)	4 バイト加算 ＋コンディションコード反映
32		int ovfsubc(char,char,char*)	1 バイト減算 ＋コンディションコード反映
33		int ovfsubw(int,int,int*)	2 バイト減算 ＋コンディションコード反映
34		int ovfsubl(long,long,long*)	4 バイト減算 ＋コンディションコード反映
35		int ovfshalc(char,char*)	1 バイト左シフト ＋コンディションコード反映

No.	項　目	仕　様	機　能
36	コンディションコード反映演算	int ovfshalw(int,int*)	2バイト左シフト＋コンディションコード反映
37		int ovfshall(long,long*)	4バイト左シフト＋コンディションコード反映
38		int ovfnegc(char,char*)	1バイト符号反転＋コンディションコード反映
39		int ovfnegw(int,int*)	2バイト符号反転＋コンディションコード反映
40		int ovfnegl(long,long*)	4バイト符号反転＋コンディションコード反映
41	10進演算	void dadd(unsigned char,char*, char*,char*)	10進加算
42		void dsub(unsigned char,char*, char*,char*)	10進減算

(1) 組み込み関数の使用方法

組み込み関数は、通常の関数と同様に関数呼び出し形式で記述します。ただし、組み込み関数を使用する場合は、必ず#include <machine. h>を宣言してください。

(2) コンディションコードレジスタの設定・参照

コンディションコードレジスタの設定・参照の組み込み関数の使用例は次の通りです。

```
#include <machine.h>
main()
{
    set_imask_ccr(0);
        :
}
```

(a) set_imask_ccr 関数

呼び出し手順：

```
#include <machine.h>
void set_imask_ccr(unsigned char mask);
```

説明：

コンディションコードレジスタ（CCR）の割り込みマスクビット（I）に mask 値（0 または 1）を設定します。

(b) get_imask_ccr 関数

呼び出し手順：

```
#include <machine.h>
unsigned char get_imask_ccr(void);
```

説明：

コンディションコードレジスタ（CCR）の割り込みマスクビット（I）の値（0 または 1）を参照します。

(c) set_ ccr 関数

呼び出し手順：

```
#include <machine.h>
void set_ccr(unsigned char ccr);
```

説明：

コンディションコードレジスタ（CCR）に ccr の値（8 ビット）を設定します。

(d) get_ccr 関数

呼び出し手順：

```
#include <machine.h>
unsigned char get_ccr(void);
```

説明：

コンディションコードレジスタ（CCR）の値を参照します。

(e) and_ccr 関数

呼び出し手順：

```
#include <machine.h>
void and_ccr(unsigned char ccr);
```

説明：

コンディションコードレジスタ（CCR）の値と ccr の論理積を算出し、結果を CCR に設定します。

(f) or_ccr 関数

呼び出し手順：

```
#include <machine.h>
void or_ccr(unsigned char ccr);
```

説明：

コンディションコードレジスタ（CCR）の値と ccr の論理和を算出し、結果を CCR に設定します。

(g) xor_ccr 関数

呼び出し手順：

```
#include <machine.h>
void xor_ccr(unsigned char ccr);
```

説明：

コンディションコードレジスタ（CCR）の値と ccr の排他的論理和を算出し、結果を CCR に設定します。

(3) エクステンドレジスタの設定・参照

エクステンドレジスタの設定・参照の組み込み関数の使用例は次の通りです。

```
#include <machine.h>
main()
{
    set_imask_exr(0);
       :
}
```

(a) set_imask_exr 関数

呼び出し手順：

```
#include <machine.h>
void set_imask_exr(unsigned char mask);
```

説明：

エクステンドレジスタ（EXR）の割り込みマスクビット（I2～I0）に mask 値（0～7）を設定します。

本関数は、CPU／動作モードが 2600a、2000a、2600n、2000n のときのみ有効です。

(b) get_imask_exr 関数

呼び出し手順：

```
#include <machine.h>
unsigned char set_imask_exr(void);
```

説明：

エクステンドレジスタ（EXR）の割り込みマスクビット（I2～I0）の値（0～7）を参照します。

本関数は、CPU／動作モードが 2600a、2000a、2600n、2000n のときのみ有効です。

(c) set_exr 関数

呼び出し手順：

```
#include <machine.h>
void set_exr(unsigned char exr);
```

説明：

エクステンドレジスタ（EXR）に exr の値（8 ビット）を設定します。
本関数は、CPU／動作モードが 2600a、2000a、2600n、2000n のときのみ有効です。

(d) get_exr 関数

呼び出し手順：

```
#include <machine.h>
unsigned char get_exr(void);
```

説明：

エクステンドレジスタ（EXR）の値を参照します。
本関数は、CPU／動作モードが 2600a、2000a、2600n、2000n のときのみ有効です。

(e) and_exr 関数

呼び出し手順：

```
#include <machine.h>
void and_exr(unsigned char exr);
```

説明：

エクステンドレジスタ（EXR）の値と exr の論理積を算出し、結果を EXR に設定します。
本関数は、CPU／動作モードが 2600a、2000a、2600n、2000n のときのみ有効です。

(f) or_exr 関数

呼び出し手順：

```
#include <machine.h>
void or_exr(unsigned char exr);
```

説明：

エクステンドレジスタ（EXR）の値と exr の論理和を算出し、結果を EXR に設定します。
本関数は、CPU／動作モードが 2600a、2000a、2600n、2000n のときのみ有効です。

(g) xor_exr 関数

呼び出し手順：

```
#include <machine.h>
void xor_exr(unsigned char exr);
```

説明：

エクステンドレジスタ（EXR）の値と exr の排他的論理和を EXR に設定します。

本関数は、CPU／動作モードが 2600a、2000a、2600n、2000n のときのみ有効です。

(4) 特殊命令

特殊命令の組み込み関数の使用例は、次のとおりです。

```
#include <machine.h>
f()
{
        :
   trapa(0);
}
```

(a) trapa 関数

呼び出し手順：

```
#include <machine.h>
void trapa(unsigned int trap_no);
```

説明：

無条件トラップの TRAPA #trap_no 命令に展開します。trap_no は 0～3 の定数です。

また、本関数は CPU／動作モードが 300 以外のときのみ有効です。

(b) sleep 関数

呼び出し手順：

```
#include <machine.h>
void sleep(void);
```

説明：

低消費電力状態命令 SLEEP に展開します。

(c) movfpe 関数

呼び出し手順：

```
#include <machine.h>
void movfpe(char*addr,char data);
```

説明：

E クロック同期データ転送命令 MOVFPE に展開します。具体的には、addr の内容を E クロックに同期したタイミングで data へ取り出します。

(d) movtpe 関数

呼び出し手順：

```
#include <machine.h>
void movtpe(char data,char*addr);
```

説明：

E クロック同期データ転送命令 MOVTPE に展開します。具体的には、data を E クロックに同期したタイミングで addr へ設定します。

(e) tas 関数

呼び出し手順：

```
#include <machine.h>
void tas(char*addr);
```

説明：

テスト・アンド・セット命令 TAS に展開します。
具体的には、addr の内容を 0 と比較し、その結果をコンディションコードレジスタ（CCR）に設定した後、addr の内容の最上位ビットを 1 にします。
本関数は、CPU／動作モードが 2600a、2000a、2600n、2000n でのみ有効です。

(f) eepmov 関数

呼び出し手順：

```
#include <machine.h>
void eepmov(char*dst,char*src,unsigned char size);
```

または、

```
void eepmov(char*dst,char*src,unsigned int size);
```

説明：

ブロック転送命令 EEPMOV に展開します。

具体的には、src で示されるアドレスから size で示されるバイト数分 dst で示すアドレスへブロック転送します。

size には、必ず定数値を指定してください。

CPU／動作モードが 300 のとき、size の値は最大 255 の制限があります。

CPU／動作モードが 300 以外のとき、size の値は最大 65535 の制限があります。ただし、256～65535 のときは、EEPMOV.W に展開されますので NMI 割り込みに注意する必要があります。

(g) mac、macl 関数

呼び出し手順：

```
#include <machine.h>
long mac(long val,int*ptr1,int*ptr2,unsigned long count);
long macl(long val,int*ptr1,int*ptr2,unsigned long count,
unsigned long mask);
```

説明：

積和演算の MAC 命令に展開します。

mac 関数は、val を MAC レジスタの初期値とします。次に、ptr1 と ptr2 で示される 2 バイトのデータを符号付きで乗算し、結果の 4 バイトデータを MAC レジスタに加算後、ptr1 と ptr2 の内容をともに＋2 します。これを count 回数繰り返します。

macl 関数は、ptr2 を繰り返し使用するために、mask との論理積演算を行ないます。

mac、macl 関数は、CPU／動作モードが 2600a、2600n のときのみ有効です。

■注意：

macl の ptr2 の指すテーブルは、mask 値の補数の 2 倍の値に境界調整されていなければなりません。下記使用例の場合、ptr2 が 8 の倍数のアドレスに割り付けられていることを、リンケージマップで確認してください。

使用例：

```
#include <machine.h>
int ptr1[10]={0,1,2,3,4,5,6,7,8,9};
int ptr2[10]={9,8,7,6,5,4,3,2,1,0};
long l1,l2;
                :
l1=mac(100,ptr1,ptr2,4);
/* l1=100+0*9+1*8+2*7+3*6 */
l2=macl(100,ptr1,ptr2,4,-4);
/* l2=100+0*9+1*8+2*9+3*8 */
```

（h）nop 関数

呼び出し手順：

```
#include <machine.h>
void nop(void);
```

説明：

nop 命令に展開します。

（5）ローテート演算

ローテート演算の組み込み関数の使用例は次のとおりです。

```
#include <machine.h>
int i,data;
f()
{
   i=rot1w(5,data);
}
```

（a）rotlc、rotlw、rotll 関数

呼び出し手順：

```
#include <machine.h>
char rotlc(int count,char data);
int rotlw(int count, int data);
long rotll(int count, long data);
```

説明：

rotlc、rotlw、rotll 関数は、それぞれ 1 バイト、2 バイト、4 バイトの data を、左方向に count ビットローテート（回転）し、その結果を返します。

（b）rotrc、rotrw、rotrl 関数

呼び出し手順：

```
#include <machine.h>
char rotrc(int count, char data);
int rotrw(int count,int data);
long rotrl(int count, long data);
```

説明：

rotrc、rotrw、rotrl 関数は、それぞれ 1 バイト、2 バイト、4 バイトの data を、右方向に count ビットローテート（回転）し、その結果を返します。

（6）コンディションコード反映演算

コンディションコード反映演算の組み込み関数の使用例は次のとおりです。

```
#include <machine.h>
int dst,src;
f()
{
if(ovfaddw(dst,src,0))
          :
else
          :
}
```

（a）ovfaddc、ovfaddw、ovfaddl 関数

呼び出し手順：

```
#include <machine.h>
int ovfaddc(char dst,char src,char * rst);
int ovfaddw(int dst,int src,int * rst);
int ovfaddl(long dst, long src,long * rst);
```

説明：

ovfaddc、ovfaddw、ovfaddl 関数は、それぞれ 1 バイト、2 バイトおよび 4 バイト同士の dst と src の加算を行ない、rst が 0 でない場合のみ結果を rst の示すエリアへ格納します。加算結果がオーバフローでなければ 0 を、オーバフローのときは 0 以外の値を返します。

これらの関数は、if 文、do 文、while 文、for 文の条件を判定する式でのみ指定することが可能です。

また、ovfaddl 関数は、cpu／動作モードが 300 以外のときのみ有効です。

（b）ovfsubc、ovfsubw、ovfsubl 関数

呼び出し手順：

```
#include <machine.h>
int ovfsubc(char dst,char src,char * rst);
int ovfsubw(int dst,int src,int * rst);
int ovfsubl(long dst, long src,long * rst);
```

説明：

ovfsubc、ovfsubw、ovfsubl 関数は、それぞれ 1 バイト、2 バイトおよび 4 バイト同士の dst と src の減算を行ない、rst が 0 でない場合のみ結果を rst の示すエリアへ格納します。減算結果がオーバフローでなければ 0 を、オーバフローのときは 0 以外の値を返します。

これらの関数は、if 文、do 文、while 文、for 文の条件を判定する式でのみ指定することが可能です。

また、ovfsubl 関数は、cpu／動作モードが 300 以外のときのみ有効です。

（c）ovfshalc、ovfshalw、ovfshall 関数

呼び出し手順：

```
int ovfshalc(char dst,char * rst);
int ovfshalw(int dst,int * rst);
int ovfshall(long dst,long *rst);
```

説明：

ovfshalc、ovfshalw および ovfshall 関数は、1 バイト、2 バイト、4 バイトの dst を左方向へ算術的 1 ビットシフトし、rst が 0 でない場合のみ結果を rst の示すエリアへ格納します。その後、算術シフト結果がオーバフローでなければ 0 を、オーバフローのときは 0 以外の値を返します。

これらの関数は、if 文、do 文、while 文、for 文の条件を判定する式でのみ指定することができます。

また、ovfshalw および ovfshall 関数は、cpu／動作モードが 300 以外のときのみ有効です。

(d) ovfnegc、ovfnegw、ovfnegl 関数

呼び出し手順：

```
int ovfnegc(char dst, char * rst);
int ovfnegw(int dst, int * rst);
int ovfnegl(long dst, long *rst);
```

説明：

ovfnegc、ovfnegw および ovfnegl 関数は、1 バイト、2 バイト、4 バイトの dst の 2 の補数を算出し、rst が 0 でない場合のみ結果を rst の示すエリアへ格納します。その後、2 の補数の結果がオーバフローでなければ 0 を、オーバフローのときは 0 以外の値を返します。

これらの関数は、if 文、do 文、while 文、for 文の条件を判定する式でのみ指定することができます。

また、ovfnegw および ovfnegl 関数は、cpu／動作モードが 300 以外のときのみ有効です。

(7) 10 進演算

10 進演算の組み込み関数の使用例は次のとおりです。

```
#include <machine.h>
char ptr1[10]={0,1,2,3,4,5,6,7,8,9};
char ptr2[10]={0,1,2,3,4,5,6,7,8,9};
char rst[10];

dadd((char)10,ptr1,ptr2,rst);
/* rst=0x0,0x2,0x4,0x6,0x8,0x10,0x12,0x14,0x16,0x18 */
```

(a) dadd 関数

呼び出し手順：

```
#include <machine.h>
void dadd(unsigned char size,char * ptr1,char * ptr2,
char * rst);
```

説明：

ptr1 から始まる size バイトのデータと、ptr2 から始まる size バイトのデータの 10 進加算を行ない、結果を rst から始まる size バイトのエリアへ格納します。

size は 1～255 の定数です。

（b）dsub 関数

呼び出し手順：

```
#include <machine.h>
void dsub(unsigned char size,char * ptrl,char * ptr2,
char * rst);
```

説明：

ptr1 から始まる size バイトのデータと、ptr2 から始まる size バイトのデータの 10 進減算を行ない、結果を rst から始まる size バイトのエリアへ格納します。

size は 1～255 の定数です。

2.3.3 簡易入出力関数のサポート

浮動小数点変換（%f,%e,%E,%g,%G）をサポートしない、簡易入出力関数を提供します。浮動小数点変換を必要としないファイル入出力を行う場合、ROM サイズを削減することができます。

（1）対象関数

fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf

（2）使用方法

#include <stdio.h>の指定行の前に、#include <no_float.h>をマクロ宣言します。

■例

```
#include <no_float.h>
 #include <stdio.h>
 void main(void)
 {
    printf(“Hello¥n”);
 }
```

■注意

#include <no_float.h>を指定したときに、上記対象関数で浮動小数点数を指定した場合、実行時の動作は保証しません。

2.4 プログラム作成上の注意事項

本節では、C コンパイラにおけるコーディング上の注意事項と、コンパイルからデバッグまでのプログラム開発上の注意事項を述べます。

2.4.1 コーディング上の注意事項

(1) float 型引数の関数

float 型の引数を宣言している関数は、必ず、原型宣言を行なうか、引数の宣言の float 型を double 型に変更してください。原型宣言を行なわず float 型引数を持つ関数を呼び出した場合、動作は保証しません。

■例

```
void f(float);―――― [1]

g( )
{
   float a;
    f(a);
}

void
f(float x)
{
    :
}
```

関数「f」は、float 型の引数を持つ関数です。この場合は、必ず［1］に示すように原型宣言を行なってください。

(2) C 言語マニュアルで評価順序を規定していない式

C 言語マニュアルで評価順序を規定していない式で、評価順序で結果が変わるようなコーディングをした場合、その動作は保証しません。

■例

`a[i]=a[++i];`	代入式の右辺を先に評価するか後に評価するかで、左辺の i の値が変わります。
`sub(++i, i);`	関数の第 1 引数を先に評価するか後に評価するかで第 2 引数の i の値が変わります。

(3) オーバフロー演算、ゼロ除算

オーバフロー演算やゼロ除算があっても、エラーメッセージを出力しません。ただし、一つの定数または定数どうしの演算で、オーバフロー演算やゼロ除算があれば、コンパイル時にエラーメッセージを出力します。

■例

```
main()
{
     int ia;
     int ib;
     float fa;
     float fb;

     ib=32767;
     fb=3.4e+38f;

/*定数または定数どうしの演算時はオーバフロー、0除算に対する  */
/*                  コンパイルエラーメッセージを出力します */

     ia=99999999999;  /*（W）定数のオーバフローを検出します*/
     fa=3.5e+40f;     /*（W）浮動小数点演算のオーバフローを検出します*/
     ia=1/0;          /*（E）0除算を検出します*/
     fa=1.0/0.0;      /*（W）浮動小数点の0除算を検出します*/

/* 実行時のオーバフローに対するエラーメッセージは出力しません*/

     ib=ib+32767;     /* 演算結果のオーバフローを無視します*/
     fb=fb+3.4e+38f ; /* 浮動小数点演算結果のオーバフローを無視します*/
     }
```

■注意

cpuexpand オプションを指定した場合、オーバフロー、アンダフローのエラーは出力しません。

（4）const 型変数への書き込み

const 型の変数を宣言していても、型変換で const 型でない型に変換して代入した場合や、分割コンパイルしたプログラムの間で、型を統一して扱っていない場合は、const 型変数への書き込みを C コンパイラでチェックできませんので、注意が必要です。

■例

```
［1］const char *p ;   /* ライブラリ関数 strcat の第１引数は char 型 */
       :               /* へのポインタ型なので、 */
   strcat(p,"abc") ; /* 引数の指す領域が書き換わることがあります。*/
```

［2］ファイル１

```
const int i;
```

ファイル２

```
extern int i;       /* 変数 i は、ファイル２では const 型で宣言して*/
    :               /* いませんのでファイル２の中で*/
  i=10;             /* 書き込んでもエラーになりません。*/
```

2.4.2 プログラム開発上の注意事項

C プログラムの作成からデバッグまでのプログラム開発上の注意事項を示します。

（1）CPU／動作モードの選択に関する注意事項

（a）コンパイル時に指定する CPU／動作モードは統一してください。

コンパイル時に cpu オプションを用いて指定する CPU／動作モードは、必ず統一してください。

異なった CPU／動作モードで作成したオブジェクトプログラムを一緒にリンクした場合、オブジェクトプログラム実行時の動作は保証しません。

（b）アセンブル時はコンパイル時の CPU／動作モードと同じ CPU 種類を指定してください。

C コンパイラが生成したアセンブリプログラムをアセンブルするとき、コンパイル時に指定した CPU／動作モードと同じ CPUタイプを cpuオプションで指定してください。

（c）リンク時には CPU／動作モードに合わせた標準ライブラリをリンクしてください。

C コンパイラが提供する標準ライブラリには 10 種類のライブラリがあります。必ず CPU／動作モードに対応するライブラリを指定してください。それ以外のライブラリを指定した場合の動作は保証しません。

参照：「1.3.5 標準ライブラリとの対応」

(2) ビット操作命令に関する注意事項

本コンパイラは、BSET、BCLR、BNOT、BST、BIST の各ビット操作命令を生成します。これらの命令は、バイト単位でデータを読み込み、ビット操作後に再びバイト単位でデータを書き込みます。一方、CPU は、ライト専用レジスタを読み込むと、レジスタの内容に関係なく 0xFF のデータを取り込みます。このため、ライト専用レジスタのビット操作命令では、操作するビット以外のビットの内容が変化してしまう場合があります。以下にライト専用レジスタに対する操作例を示します。

■例

インクルードファイル(300x.h)の内容

```
struct S_p4ddr{
        unsigned char p7:1;
                :
        unsigned char p0:1;
};
union SS{
        unsigned char Schar;
        struct S_p4ddr Sstr;
};
#define P4DDR (*(union SS *)0xffffc5)
#define P0 0x1
```

Ｃソースプログラムの内容

```
#include "300x.h"
sub()
{
  unsigned char DDR=P4DDR.Schar;
  DDR &=~P0;
  P4DDR.Schar=DDR;
}
```

(3) トラブル発生時の対処方法

表 2.12　トラブル発生時の対処方法（1）

No.	現　　象	確　認　内　容	対　処　方　法	参　　照
1	C コンパイラ出力のオブジェクトプログラムにアセンブラ埋め込み部分が出力されない。	コンパイラオプションで code＝asmcode を指定したか。	コンパイル時に、code＝asmcode を指定してください。	「2.3.1 (2) アセンブラ埋め込み機能」
2	リンク時に、エラー No.314 cannot found section が出力される。	リンケージエディタの start オプションにおいて、コンパイラ出力のセクション名は大文字で指定しているか。	正しいセクション名を指定してください。	「2.2.1 オブジェクトプログラムの構造」 「1.3.3 コンパイラオプション」

No.	現　象	確 認 内 容	対 処 方 法	参　照
3	リンク時に、エラー No.105 undefined external symbol が出力される。	C プログラムとアセンブリプログラム間で変数を相互参照している場合、アセンブリプログラム内で下線を付加しているか。	正しい変数名で参照してください。	「2.2.3（1）外部名の相互参照方法」
		C プログラムで C ライブラリ関数を使用していないか。	リンク時に入力ライブラリとして標準ライブラリを指定してください。	標準ライブラリとの対応：「1.3.5」
		未定義参照シンボル名が＄で始まっていないか。（標準ライブラリ中の実行時ルーチンを使用しています）		標準ライブラリ中の実行時ルーチン：「3.2.1（2）注意」
		C ライブラリ関数の標準入出力ライブラリを使用していないか。	低水準インタフェースルーチンを作成してリンクしてください。	「3.4（6）低水準インタフェースルーチン」
4	リンク時に、エラー No.108 relocation size overflow が出力される。	abs8、abs16 オプションを指定しているか。または、#pragma abs8、#pragma abs16 を使用しているか。	8 ビット、16 ビット短絶対アドレス領域を正しい領域に割り付けてください。	「1.3.3 コンパイラオプション」 「2.3.1（1）短絶対アドレス機能」
5	デバッグ時に C ソースレベルのデバッグができない。	コンパイル、アセンブル、リンク時とも debug オプションを指定したか。	コンパイル時、アセンブル時、およびリンク時のそれぞれに debug オプションを指定してください。	「1.3.3 コンパイラオプション」

3. システム組み込み

第３章　目次

3.1. システム組み込みの概要 103
3.2. メモリ領域の割り付け 104
3.2.1 静的領域の割り付け 104
3.2.2 動的領域の割り付け 108
3.3 実行環境の設定 113
3.4 C ライブラリ関数の実行環境の設定 120

3.1. システム組み込みの概要

本章では、C プログラムを H8S/2600、H8S/2000、H8/300H または H8/300 を応用したシステムに組み込む方法を説明します。

C プログラムをシステムに組み込むには、以下の準備が必要です。

(1) メモリの割付け

C プログラムの各セクション、スタック領域、ヒープ領域を、システム上の ROM、RAM のメモリ領域に割り当てる必要があります。

(2) C プログラム実行環境の設定

C プログラムの実行環境を設定する処理には、レジスタの初期設定、メモリ領域の初期化、C プログラムの起動があります。これらの機能をアセンブリプログラムで実現する必要があります。

また、入出力等の C ライブラリ関数をご使用になる場合は、実行環境の設定時にライブラリの初期化をする必要があります。特に入出力（stdio.h）とメモリ割り付け（stdlib.h）の機能をご使用になる場合は、システムごとに、低水準の入出力ルーチンやメモリ割り付けルーチンを作成する必要があります。

第 2 節では C プログラムのメモリ領域のアドレスを決定する考え方を説明し、実際にアドレスを決定するためのリンケージエディタのコマンドの指定方法について実例を挙げて説明します。

第 3 節では実行環境設定の項目を説明し、設定プログラムの実例について説明します。

第 4 節では C ライブラリ関数の初期設定処理と、低水準ルーチンの作成方法を説明します。

3.2. メモリ領域の割り付け

C コンパイラの出力したオブジェクトプログラムをシステムに組み込むためには、プログラムの使用するメモリ領域のサイズを決定し、それぞれの領域を適切なメモリアドレスに割り付ける必要があります。

C プログラムが使用するメモリ領域には、C プログラム中の関数に対応する機械語や外部データ定義で宣言したデータ領域のように静的に割り付ける領域と、スタック領域のように動的に割り付ける領域があります。

以下、各領域の割り付け方を説明します。

3.2.1 静的領域の割り付け

(1) 静的領域の内容

オブジェクトプログラムの各セクション（プログラム領域、定数領域、初期化データ領域、未初期化データ領域）は静的領域に割り付けます。

(2) サイズの算出法

静的領域のサイズは、C コンパイラが生成するオブジェクトプログラムサイズと C プログラムが使用するライブラリ関数のサイズの合計になります。オブジェクトプログラムをリンクしたあと、リンケージマップリストにライブラリを含めた各セクションのサイズを出力しますので、静的領域のサイズを知ることができます。

リンクまえに静的領域のサイズを概算する場合は、コンパイルリストの統計情報にセクションごとのサイズを出力しますので、これに基づいてサイズを算出することができます。**図 3.1** に統計情報の例を示します。

```
******* SECTION SIZE INFORMATION *******

PROGRAM  SECTION(P):                                0x00000080 Byte(s)
CONSTANT SECTION(C):                                0x00000004 Byte(s)
DATA     SECTION(D):                                0x00000004 Byte(s)
BSS      SECTION(B):                                0x00000004 Byte(s)

TOTAL PROGRAM  SECTION: 0x00000080 Byte(s)
TOTAL CONSTANT SECTION: 0x00000004 Byte(s)
TOTAL DATA     SECTION: 0x00000004 Byte(s)
TOTAL BSS      SECTION: 0x00000004 Byte(s)

    TOTAL PROGRAM SIZE: 0x0000008C Byte(s)
```

図 3.1　**統計情報例**

標準ライブラリを使用しない場合は、コンパイル単位ごとに出力する統計情報のセクションごとのサイズの合計が静的領域のサイズになります。

また、標準ライブラリを使用している場合、各セクションのメモリ領域サイズにはライブラリ関数の使用するメモリ領域サイズを加えなければなりません。静的領域サイズは、必ずリンケージマップリストで確認してください。標準ライブラリ関数には、C 言語仕様で規定したC ライブラリ関数の他に、C プログラムを実行する上で必要な算術演算を行なうルーチンも含まれています。そのため、C ソースプログラム上でライブラリ関数の使用を指定しなくても、標準ライブラリが必要な場合がありますので注意してください。

■注意

C コンパイラが提供する標準ライブラリの中には、C 言語仕様で規定したC ライブラリ関数と、C プログラムを実行する上で必要な算術演算を行なうルーチン（実行時ルーチン）を含みます。

実行時ルーチンのサイズも C ライブラリ関数と同じようにメモリ領域サイズに加える必要があります。

C プログラムで使用する実行時ルーチンは、C コンパイラが出力するコンパイルリストのシンボル割り付け情報から知ることができます。

以下に具体例を示します。

■例

Cプログラム

```
long a,b;
main()
{
        a *= b;
}
```

Cコンパイラ出力のシンボル割り付け情報

```
******* STACK FRAME INFORMATION *******

FILE NAME:  main.c

Function (File main.c   , Line     2): main

Parameter Area Size     : 0x00000000 Byte(s)
Linkage Area Size       : 0x00000008 Byte(s)
Local Variable Size     : 0x00000000 Byte(s)
Temporary Size          : 0x00000000 Byte(s)
Register Save Area Size : 0x00000000 Byte(s)
Total Frame Size        : 0x00000008 Byte(s)

Used Runtime Library Name
$MULL$3                                 ;実行時ルーチン
```

（3）ROM、RAM の割り付け

プログラムを ROM 化する場合は、静的な領域を以下のように ROM と RAM に分けて割り付けます。

プログラム領域　　（セクション P）→　ROM
定数領域　　（セクション C）→　ROM
未初期化データ領域（セクション B）→　RAM
初期化データ領域　（セクション D）→　ROM、RAM（下記（4）参照）

（4）初期化データ領域の割り付け

初期化データ領域は、初期値を持ったデータ領域ですが、値の変更が可能なので、リンク時には ROM 上に置き、プログラムの実行開始時に RAM 上にコピーします。したがって、初期化データの領域については、ROM 上と RAM 上に、二重に領域をとる必要があります。

ただし、初期値を指定した静的変数を変更しないようにプログラムを作成すれば、初期化データの領域は ROM 上に置くだけでよく、二重に割り付ける必要はありません。

(5) メモリ割り付け例とリンク時のアドレス指定方法

アブソリュートロードモジュール作成時に、リンケージエディタのオプションまたはサブコマンドで各セクションごとに割り付ける領域のアドレスを指定します。以下、静的領域のメモリ割り付け例とリンク時の指定方法について説明します。

図 3.2 に、H8S/2600 アドバンストモードにおける静的な領域の割り付け例を示します。

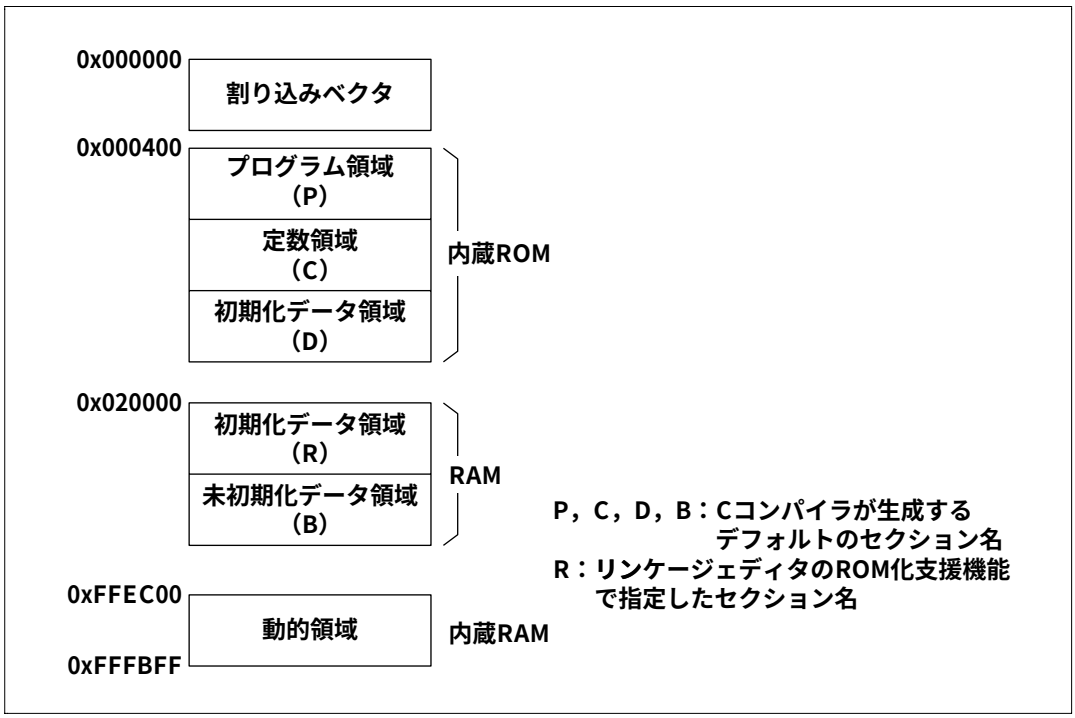


図 3.2　**静的な領域の割り付け例**

図 3.2 に示すメモリ割り付けを行なう場合、リンク時に以下のサブコマンドを指定します。

```
    :
ROM△(D,R)                                ……… [1]
START△P,C,D(400),R,B(20000)              ……… [2]
    :
```

■説明

[1] セクション名 D と同じ大きさのセクション R を出力ロードモジュールに確保します。また、セクション D に割り付けられたシンボルを参照している場合、セクション R 上のアドレスとなるようリロケーションします。
セクション D は ROM 上、セクション R は RAM 上の初期化データセクション名になります。

[2] セクション P、C、D を内蔵 ROM のアドレス 0x400 から連続した領域に割り付けます。また、セクション R、B を RAM のアドレス 0x20000 から連続したアドレスに割り付けます。

■注意

リンケージエディタのサブコマンド ROM は、リンケージエディタ（Ver. 5.0）以上で使用することができます。

3.2.2 動的領域の割り付け

(1) 動的領域の内容

C プログラムで使用する動的領域には、以下の二つがあります。

(a) スタック領域
(b) ヒープ領域（メモリ割り付けライブラリ関数で使用）

(2) サイズの算出法

(a) スタック領域

C プログラムの使用するスタック領域は、関数呼び出しのたびにスタック上に割り付け、関数のリターン時に解放します。スタック領域のサイズを算出するためには、まず各関数ごとのスタック使用量を算出し、関数の呼び出し関係から実際のスタック使用量を算出します。

各関数の使用するスタック領域

各関数の使用するスタック領域は、コンパイルリストのシンボル割り付け情報（Total Frame Size）から知ることができます。

■例

次の C プログラムに対するシンボル割り付け情報とスタック使用量の算出法を示します。ここでは、H8S/2600 用アドバンストモードの場合について説明します。

```
extern int h(char, char *, double);
int
h(char a, register char *b, double c)
{
    char   *d;

    d= &a;
    h(*d,b,c);
    {
       register int i;

       i= *d;
       return i;
     }
}
```

```
******* STACK FRAME INFORMATION ********

FILE NAME: m0280.c

Function (File m0280.c   , Line     3): h

  Parameter Allocation
    a                                   0xfffffff7    saved from R0L
    b                                REG  ER5           saved from ER1
    c                                   0x00000008

  Level 1(File m0280.c , Line        4)  Automatic/Register Variable Allocation
    d                                   0xfffffff2

  Leve1 2(File m0280.c , Line        9)  Automatic/Register Variable Allocation
    i                                REG  R4

Parameter Area Size          : 0x00000008 Byte(s)
Linkage Area Size            : 0x00000008 Byte(s)
Local Variable Size          : 0x00000006 Byte(s)
Temporary Size               : 0x00000000 Byte(s)
Register Save Area Size      : 0x00000008 Byte(s)
Total Frame Size             : 0x0000001e Byte(s)
```

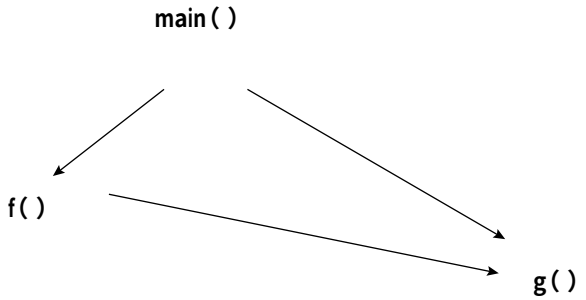
関数の使用するスタック領域サイズは、Total Frame Size の 0xle つまり 30 バイトとなります。

スタック使用量の算出法

関数の呼び出し関係から使用するスタック領域のサイズを算出します。

■例

関数の呼び出し関係と各関数のスタック使用量の例を**図** 3.3 に示します。



関数名	スタック使用量（バイト）
main	30
f	32
g	24

図 3.3　**関数呼び出しの関係とスタック使用量の例**

この場合、関数 f を介して関数 g が呼ばれた時のスタック領域のサイズは、**表** 3.1 によって計算します。

表 3.1　**スタックサイズの計算例**

呼出し経路	スタックサイズ計	
main（30） → f（32） → g（24）	86	**←スタック使用量**
main（30） → g（24）	54	**（最大値）**

このように、呼び出しレベルの一番深い関数についてスタック領域のサイズを計算し、その最大値（この場合 86 バイト）のスタック領域を最低限割り付ける必要があります。

標準ライブラリのライブラリ関数を使用する場合には、ライブラリ関数を含めたスタック領域のサイズを計算する必要があります。ライブラリ関数の使用するスタック領域のサイズについては、製品添付の**「標準ライブラリのスタック使用量一覧」**を参照してください。

■注意

C プログラムの中で再帰呼び出しを行なっている場合は、再帰的に呼び出す回数の最大値を算出してから、その関数のスタック領域のサイズに再帰的に呼び出す回数を掛けて計算してください。

（b）ヒープ領域

ヒープ領域で使用するメモリ領域のサイズは、C プログラム内でメモリ管理ライブラリ関数（calloc、malloc、realloc 関数）によって割り付ける領域の合計です。ただし、メモリ管理ライブラリ関数は、1 回の呼び出しのたびに管理用の領域として 2 バイト（cpu＝2600n、cpu＝2000n、cpu＝300hn、cpu＝300 指定時）または 4 バイト（cpu＝2600a、cpu＝2000a、cpu＝300ha 指定時）を使用しますので、実際に確保する領域サイズにこの管理領域のサイズを加えて計算してください。

また、C コンパイラはヒープ領域をユーザ指定のメモリサイズ（_sbrk_size）の単位で管理しています。_sbrk_size の指定方法は**「3.4（4）C ライブラリ関数の初期設定」**を参照してください。ヒープ領域として確保する領域サイズ（HEAPSIZE）は次のように計算してください。

HEAPSIZE＝_sbrk_size×n（n≧1）

（メモリ管理ライブラリによって割り付ける領域サイズ）＋管理領域サイズ≦HEAPSIZE

入出力ライブラリ関数は、内部処理の中でメモリ管理ライブラリ関数を使用しています。入出力の中で割り付ける領域のサイズは、

cpu＝2600n、2000n、300hn、300 指定時

514 バイト×（同時にオープンするファイルの数の最大値）

cpu＝2600a、2000a、300ha 指定時

516 バイト×（同時にオープンするファイルの数の最大値）

になります。

■注意

メモリ管理ライブラリ関数の free 関数で解放した領域は、再びメモリ管理ライブラリ関数で領域を確保するときに再利用しますが、割り付けを繰り返すことによって空き領域のサイズの合計は十分でも、空き領域が小さな領域に分割しているために、新たに要求した大きなサイズの領域を確保できないという状況が生じることがあります。このような状況を避けるために、以下の注意に従ってヒープ領域を使用してください。

・サイズの大きな領域は、なるべくプログラムの実行開始直後に確保してください。
・解放して再利用するデータ領域のサイズをなるべく一定にしてください。

（3）動的領域の割り付け方

動的領域はRAM上に割り付けます。

スタック領域については、プログラム起動時の初期設定（INIT）でスタック領域の最上位アドレスをSP（スタックポインタ）に設定することにより割り付ける場所が決まります。

ヒープ領域については、低水準インタフェースルーチン（sbrk）の初期設定で割り付ける場所が決まります。

それぞれ、**「3.3（2）初期設定（INIT）」**、**「3.4（6）低水準インタフェースルーチン」**を参照してください。

3.3 実行環境の設定

本節では、C プログラムの実行に必要な環境を設定するための処理について説明します。ただし、C プログラムを実行する環境はユーザシステムごとに異なりますので、ご使用になっているシステムの仕様に合わせて実行環境の設定プログラムを作成する必要があります。

ここでは、プログラムの実行環境の最も基本的な構成として、C ライブラリ関数を使用しない場合について説明します。

C ライブラリ関数を使用する場合は、**「3.4 C ライブラリ関数の実行環境の設定」**を参照してください。

図 3.4 にプログラムの構成例を示します。

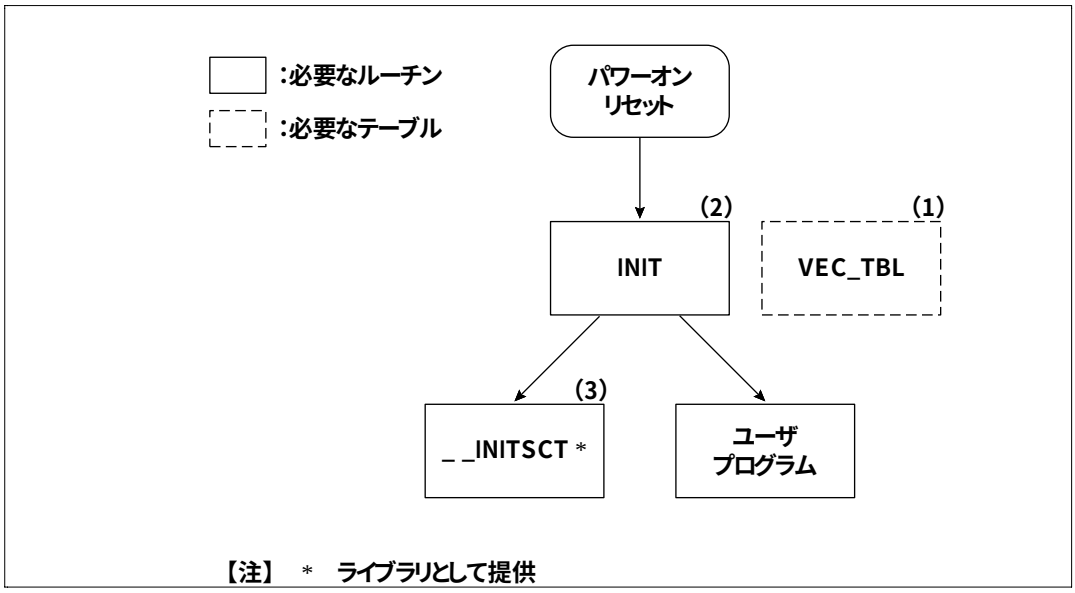


図 3.4 プログラムの構成例（C ライブラリ関数を使用しない場合）

各構成ルーチンの内容は以下のとおりです。

(1) ベクタテーブルの設定（VEC_TBL）

パワーオンリセットでレジスタの初期設定プログラム（INIT）が起動されるように、ベクタテーブルを設定します。

(2) 初期設定（INIT）

レジスタの初期設定を行なったあと、初期設定ルーチンを順次呼び出します。

(3) セクションの初期化（__INITSCT）

初期値が設定されていない静的変数領域（未初期化データ領域）をゼロで初期化します。また、初期化データ領域の初期値を ROM 上から RAM 上にコピーします。

本ルーチンは、標準ライブラリのライブラリ関数「__INITSCT」としても提供しています。

以下、この構成に従って各処理の実現方法について解説します。

（1）ベクタテーブルの設定（VEC_TBL）

パワーオンリセットで、レジスタの初期設定を行なう関数「INIT」が呼び出されるようにするためには、ベクタテーブルの 0 番地に関数「INIT」の先頭アドレスを設定する必要があります。

また、ユーザシステムで割り込み処理やメモリ間接関数呼び出しを使用する場合は、割り込みベクタやアドレステーブルの設定も本ルーチンで行ないます。以下に C プログラムおよびアセンブリプログラムで記述した場合のコーディング例を示します。

■例（C プログラム例）

```
#pragma interrupt IRQ0
extern void INIT(void);
extern void IRQ0(void);

#pragma section vect1       /* #pragma section 宣言により vec_table1 を       */
                            /*               Cvect1 セクションに出力します。  */
                            /* リンク時に,start オプションで Cvect1 セクション  */
                            /* を 0 番地に割り付けるよう指定します。            */

const void (*const vec_table1[])(void)={INIT};
#pragma section vect2       /* #pragma section 宣言により vec_table2 を       */
                            /*               Cvect2 セクションに出力します。  */
                            /* リンク時に,start アドレスで Cvect2 セクションを  */
                            /*       所定番地に割り付けるよう指定します。       */

const void (*const vec_table2[])(void)={IRQ0};
                            /* 空いている番地に indirect オプション           */
                            /*     #pragma   indirect 指定で                  */
                            /*     作成されたアドレステーブルを割り付けます。 */
```

■例（アセンブリプログラム例）

（CPU／動作モードが 2600n、2000n、300hn、300 の場合）

```
        .EXPORT   $IRQ0
        .IMPORT   _INIT
        .IMPORT   _IRQ0
        .SECTION  Cvect1,DATA,LOCATE=H'0000
                                ;セクション制御命令で Cvect1 セクションを 0 番地に配置。
        .DATA.W   _INIT         ;「_INIT」の先頭アドレスを 0～1 番地に配置。
        .SECTION  Cvect2,DATA,LOCATE=H'0040
                                ;セクション制御命令で Cvect2 セクションを 0x40 番地に配置。
$IRQ0:  .DATA.W   _IRQ0         ;「_IRQ0」の先頭アドレスを 0x40～0x41 番地に配置。
        .END
```

（CPU／動作モードが 2600a、2000a、300ha の場合）

```
        .EXPORT   $IRQ0
        .IMPORT   _INIT
        .IMPORT   _IRQ0
        .SECTION  Cvect1,DATA,LOCATE=H'0000
                               ;セクション制御命令で Cvect1 セクションを 0 番地に配置。
        .DATA.L   _INIT        ;「_INIT」の先頭アドレスを 0～3 番地に配置。
        .SECTION  Cvect2,DATA,LOCATE=H'0080
                               ;セクション制御命令で Cvect2 セクションを 0x80 番地に配置。
$IRQ0:  .DATA.L   _IRQ0        ;「_IRQ0」の先頭アドレスを 0x80～0x83 番地に配置。
        .END
```

（2）初期設定（INIT）

ここでは、レジスタの初期設定を行ない、初期設定ルーチンを順次呼び出したあと、main 関数を呼び出します。

以下にコーディング例を示します。

■例（CPU／動作モードが 2600n、2000n、300hn、300 の場合）

```
#include <machine.h>
#pragma noregsave INIT
void main(void);
void _INITSCT(void);
void INIT(void)
#pragma asm
   MOV.W #H'FFFE,R7         ; INIT 関数の始まる前にスタックのアドレスを設定します。
#pragma endasm
{
   set_imask_ccr(0);        /* 割り込みマスクビットを 0 に設定。*/
   _INITSCT();              /* セクションの初期化ルーチン「_INITSCT」の呼び出し。*/
   main();                  /* main 関数の呼び出し。*/
   sleep();                 /* sleep 命令に展開。*/
}
```

■例（CPU／動作モードが 2600a、2000a、300ha の場合）

```
#include <machine.h>
#pragma noregsave INIT
void main(void);
void _INITSCT(void);
void INIT(void)
#pragma asm
   MOV.L #H'FFFBFE,SP       ; INIT 関数の始まる前にスタックのアドレスを設定します。
#pragma endasm
{
   set_imask_ccr(0);        /* 割り込みマスクビットを 0 に設定。*/
   _INITSCT();              /* セクションの初期化ルーチン「_INITSCT」の呼び出し。*/
   main();                  /* main 関数の呼び出し。*/
   sleep();                 /* sleep 命令に展開。*/
}
```

（3）セクションの初期化（__INITSCT）

C プログラムの実行環境を設定するために、未初期化データ領域をゼロで初期化することと、ROM 上にある初期化データを RAM 上にコピーすることが必要です。このルーチンは標準ライブラリのライブラリ関数「__INITSCT」として提供していますので、**「（2）初期設定（INIT）」**に示すように、初期設定ルーチンの中で「__INITSCT」関数を呼び出すだけで実現できます。

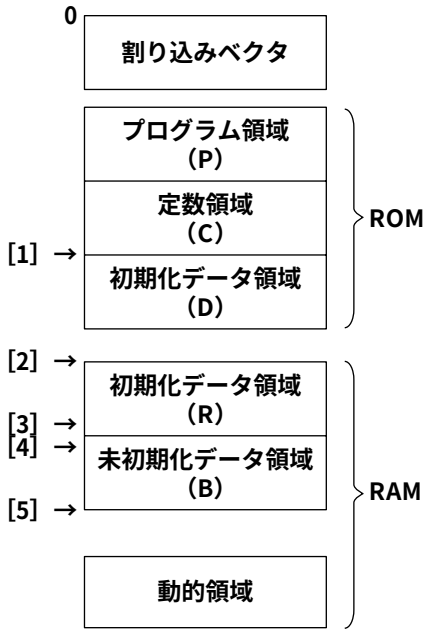
ただし、「__INITSCT」関数を使用するためには次の条件をすべて満たす必要があります。

・ 未初期値データ領域、初期化データ領域のセクション名がそれぞれ B、D であること。（セクション名 B、D は、コンパイラオプション section や#pragma section で特定の名前を指定しないときに C コンパイラが作成する名前です。）
・ abs8、abs16 オプションを使用していないこと。また、#pragma abs8、#pragma abs16 を使用していないこと。
・ 初期化データ領域の RAM 上のセクション名として R を仮定しており、ロードモジュール作成時にリンケージエディタの ROM 化支援オプション（またはサブコマンド）「ROM＝(D,R)」を指定すること。
・ リンケージエディタ（Ver.5.0）以上を使用すること。

標準ライブラリのライブラリ関数「__INITSCT」を使用しない場合は、以下の手順に従って「__INITSCT」関数を作成してください。

「__INITSCT」の処理を行なうためには、次のアドレスを知る必要があります。

・初期化データ領域の ROM 上の先頭アドレス（［1］）
・初期化データ領域の RAM 上の先頭アドレス（［2］）、最終アドレス（［3］）
・未初期化データ領域の先頭アドレス（［4］）、最終アドレス（［5］）



これらのアドレスを知るためには、次のアセンブリプログラムを作成、リンクしてください。

（H8S/2600、H8S/2000、H8/300H ノーマルモードおよび H8/300 の場合）

```
        .SECTION   D,DATA,ALIGN=2
        .SECTION   R,DATA,ALIGN=2
        .SECTION   B,DATA,ALIGN=2
        .SECTION   C,DATA,ALIGN=2

__D_ROM  .DATA.W    (STARTOF D)                ;［1］セクション D の先頭アドレス
__D_BGN  .DATA.W    (STARTOF R) *1             ;［2］セクション R の先頭アドレス
__D_END  .DATA.W    (STARTOF R) + (SIZEOF R)   ;［3］セクション R の最終アドレス
                                *2
__B_BGN  .DATA.W    (STARTOF B)                ;［4］セクション B の先頭アドレス
__B_END  .DATA.W    (STARTOF B)+(SIZEOF B)     ;［5］セクション B の最終アドレス

        .EXPORT    __D_ROM
        .EXPORT    __D_BGN
        .EXPORT    __D_END
        .EXPORT    __B_BGN
        .EXPORT    __B_END
        .END
```

（H8S/2600、H8S/2000、H8/300H アドバンストモードの場合）

```
        .SECTION   D,DATA,ALIGN=2
        .SECTION   R,DATA,ALIGN=2
        .SECTION   B,DATA,ALIGN=2
        .SECTION   C,DATA,ALIGN=2

__D_ROM  .DATA.L    (STARTOF D)                ;［1］セクション D の先頭アドレス
__D_BGN  .DATA.L    (STARTOF R) *1             ;［2］セクション R の先頭アドレス
__D_END  .DATA.L    (STARTOF R)+ (SIZEOF R)    ;［3］セクション R の最終アドレス
                                *2
__B_BGN  .DATA.L    (STARTOF B)                ;［4］セクション B の先頭アドレス
__B_END  .DATA.L    (STARTOF B) + (SIZEOF B)   ;［5］セクション B の最終アドレス

        .EXPORT    __D_ROM
        .EXPORT    __D_BGN
        .EXPORT    __D_END
        .EXPORT    __B_BGN
        .EXPORT    __B_END
        .END
```

■注意

- セクション名 B、D はコンパイラオプション section で指定した未初期化データ領域、初期化データ領域のセクション名を指定してください。
- セクション名 R は、リンク時に ROM 化支援オプション ROM で指定した RAM 上のセクション名を指定してください。
- 下線部*1、*2 は、リンケージエディタ（Ver. 5.0）以上での指定方法です。
  Ver.5.0 より前のリンケージエディタをご使用の場合は、実際の割り付けアドレスを指定する必要があります。

■例　（RAM 上の初期化データ領域を 0x8000 から割り付ける場合）

```
__D_BGN     .DATA.L   H'8000 *1                    ; [2]
__D_END     .DATA.L   H'8000 + (SIZEOF D)   *2     ; [3]
```

上記の準備をすれば、セクションの初期化ルーチンは C 言語で記述することができます。以下にプログラム例を示します。

■例

```
extern char  *_D_ROM, *_B_BGN, *_B_END, *_D_BGN, *_D_END;
extern void  _INITSCT( );

void  _INITSCT( )
{

        char *p, *q ;

        /* 未初期化データ領域をゼロで初期化 */

        for (p=_B_BGN ; p<_B_END ; p++)
              *p=0 ;

        /*初期化データを ROM 上から RAM 上へコピー */

        for (p=_D_BGN , q=_D_ROM ; p<_D_END ; p++, q++)
              *p=*q ;
}
```

3.4 C ライブラリ関数の実行環境の設定

C ライブラリ関数をご使用になる場合は、C プログラムの実行環境の設定として C ライブラリ関数の初期化をする必要があります。特に入出力（stdio.h）とメモリ割り付け（stdlib.h）の機能をご使用になる場合は、システムごとに低水準の入出力ルーチンやメモリ割り付けルーチンを作成する必要があります。

本節では、C ライブラリ関数使用時の C プログラムの実行環境を設定方法について説明します。

図 3.5 にプログラムの構成例を示します。

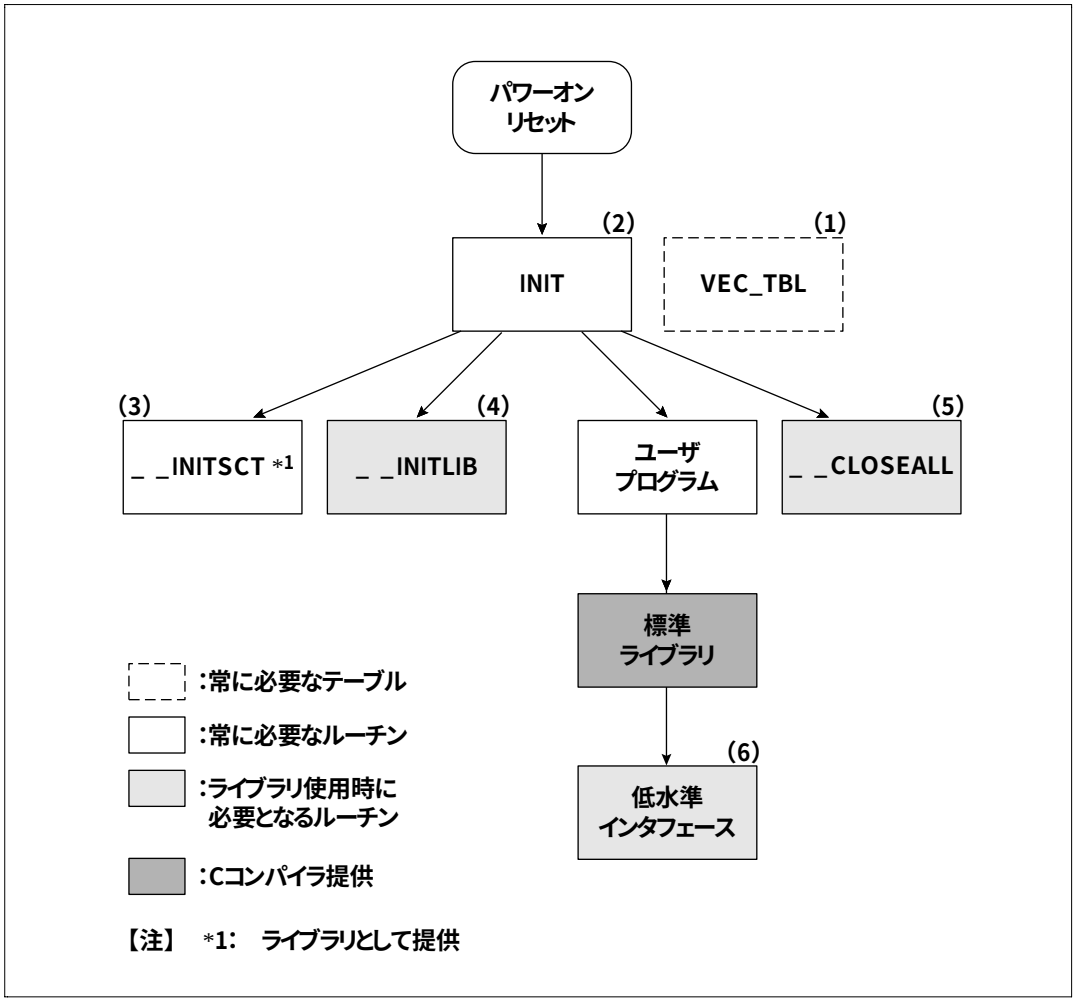


図 3.5 **プログラムの構成例（C ライブラリ関数を使用する場合）**

ライブラリ使用時に必要な各構成ルーチンの内容は以下のとおりです。

（1）ベクタテーブルの設定（VEC_TBL）
パワーオンリセットでレジスタの初期設定プログラム（INIT）が起動されるように、ベクタテーブルを設定します。
（2）初期設定（INIT）
レジスタの初期設定を行なったあと、初期設定ルーチンを順次呼び出します。
（3）セクションの初期化（__INITSCT）
初期値が設定されていない静的変数領域（未初期化データ領域）をゼロで初期化します。また、初期化データ領域の初期値を ROM 上から RAM 上にコピーします。
本ルーチンは、標準ライブラリのライブラリ関数「__INITSCT」として提供しています。
（4）C ライブラリ関数の初期設定（__INITLIB）
C ライブラリ関数の中で、初期設定の必要なものについて、初期設定を行ないます。特に、標準入出力を行なう準備をします。
（5）ファイルのクローズ（__CLOSEALL）
オープンしているファイルを全てクローズします。
（6）低水準インタフェースルーチン
標準入出力、メモリ管理ライブラリを使用する場合に必要なライブラリ関数とユーザシステムとの間のインタフェースルーチンです。

以下、この構成に従って各処理の実現方法について解説します。

■注意

プログラムの終了処理を行なう C ライブラリ関数 exit、onexit、abort 関数を使用する場合は、ユーザシステムに合わせてこれらの関数を作成する必要があります。具体的なプログラム例を**「付録 D　終了処理関数の作成例」**に示します。

なお、C ライブラリ関数 assert マクロを使用する場合、abort 関数は必ず作成する必要があります。

（1）ベクタテーブルの設定（VEC_TBL）

ベクタテーブルは C ライブラリ関数を使用しない場合と同じです。**「3.3　実行環境の設定」**を参照してください。

（2）初期設定（INIT）

C ライブラリ関数を使用する場合には、ここでライブラリの初期設定を行なう「__INITLIB」とファイルのクローズ処理を行なう「__CLOSEALL」を呼び出します。以下に「INIT」のコーディング例を示します。

■例（CPU／動作モードが 2600n、2000n、300hn、300 の場合）

```
#include <machine.h>
#pragma noregsave(INIT)      /*レジスタ退避／回復コード抑止 */
void main(void);
void _INITSCT(void);
void _INITLIB(void);
void _CLOSEALL(void);
void INIT(void)
#pragma asm
     MOV.W #H'FFFE,R7        ;INIT 関数の始まる前にスタックのアドレスを設定します
#pragma endasm
{
     set_imask_ccr(0);       /*割込みマスクビットを 0 に設定 */
     _INITSCT( );            /*セクションの初期化ルーチン「_INITSCT」の呼び出し*/
     _INITLIB( );            /*ライブラリの初期化ルーチン「_INITLIB」の呼び出し*/
     main( );                /*main 関数の呼び出し */
     _CLOSEALL( );           /*ファイルのクローズルーチン「_CLOSEALL」の呼び出し*/
     sleep( );               /*sleep 命令に展開*/
}
```

■例（CPU／動作モードが 2600a、2000a、300ha の場合）

```
#include <machine.h>
#pragma noregsave(INIT)       /* レジスタ退避／回復コード抑止 */
void main(void);
void _INITSCT(void);
void _INITLIB(void);
void _CLOSEALL(void);
void INIT(void)
#pragma asm
    MOV.L #H'FFFBFE,SP        ;INIT 関数の始まる前にスタックのアドレスを設定します
#pragma endasm
{
   set_imask_ccr(0);          /* 割込みマスクビットを 0 に設定 */
   _INITSCT( );               /* セクションの初期化ルーチン「_INITSCT」の呼び出し */
   _INITLIB( );               /* ライブラリの初期化ルーチン「_INITLIB」の呼び出し */
   main( );                   /* main 関数の呼び出し */
   _CLOSEALL( );              /* ファイルのクローズルーチン「_CLOSEALL」の呼び出し */
   sleep( );                  /* sleep 命令に展開 */
}
```

（3）セクションの初期化（__INITSCT）

セクションの初期化は C ライブラリ関数を使用しない場合と同じです。**「3.3 実行環境の設定」**を参照してください。

（4）C ライブラリ関数の初期設定（__INITLIB）

C ライブラリ関数の中には、初期設定が必要な関数があります。それらの関数を使用する場合、当然使用するまえに定められた初期設定を行なわなければなりません。本項では、プログラム起動ルーチンの中の「__INITLIB」の中で初期設定を行なう場合を例にとり説明します。

実際に使用する機能に合わせた必要最低限の初期設定を行なうために、以下の指針を参考にしてください。

- ライブラリのエラー状態を示す「errno」の初期設定はすべてのライブラリ関数共通に必要です。
- 〈stdio.h〉の各関数と assert マクロを使用する場合、標準入出力の初期設定が必要です。
- 作成した低水準インタフェースルーチンの中で初期設定が必要な場合、低水準インタフェースルーチンの仕様に合わせた初期設定が必要です。
- rand 関数、strtok 関数を使用する場合、標準入出力以外の初期設定が必要です。

ライブラリの初期設定を行なうプログラム例を以下に示します。

■例

```
#include <stdlib.h>
const size_t _sbrk_size=516 ; *1   /* ヒープ領域確保サイズの最小単位を指定 */
extern void _INIT_LOWLEVEL(void) ;
extern void _INIT_IOLIB(void) ;
extern void _INIT_OTHERLIB(void) ;

void _INITLIB(void)                /* アセンブリルーチンのシンボル名から下線を一つ削除 */
{
  errno=0;                         /* ライブラリ関数共通の初期設定 */

  _INIT_LOWLEVEL( ) ;              /* 低水準インタフェースの初期設定ルーチンの呼び出し*/
  _INIT_IOLIB( ) ;                 /* 標準入出力の初期設定ルーチンの呼び出し */
  _INIT_OTHERLIB( ) ;              /* 標準入出力以外の初期設定ルーチンの呼び出し */
}
```

以下、標準入出力の初期設定ルーチン（_INIT_IOLIB）、標準入出力以外の初期設定ルーチン（_INIT_OTHERLIB）の作成例を示します。低水準インタフェースルーチンの初期設定ルーチン（_INIT_LOWLEVEL）は、ユーザ作成の低水準インタフェースルーチンの仕様に合わせて作成してください。

【注】 *1 C **コンパイラがヒープ領域として確保するメモリサイズの最小単位を指定します。指定がない場合は、**1028 **バイト（アドバンストモード時は** 1032 **バイト）とみなします。**

一回に確保するメモリサイズは、必ず偶数を指定してください。奇数を指定した場合の動作は保証しません。

確保サイズは、（実際に使用するメモリサイズ）＋管理領域サイズを目安にしてください。管理領域サイズは、cpu=300，300hn，2000n，2600n **のとき** 4 **バイト、**cpu=300ha，2000a，2600a **のとき** 8 **バイトです。**

（a）標準入出力の初期設定ルーチン（_INIT_IOLIB）の作成例

標準入出力の初期設定では、ファイルを参照するために必要な FILE 型データ（**図** 3.6）の初期設定と標準入出力ファイルのオープンを行ないます。FILE 型データの初期設定は、必ず標準入出力ファイルのオープンの前に行なってください。

標準入出力の初期設定を行なうプログラム例を示します。

■例

```
#include <stdio.h>

#define N 4 *4
const int _nfiles =N; *4
struct _iobuf _iob[N]; *4
unsigned char sml_buf[N]; *4
void _INIT_IOLIB(void)
{
   FILE *fp ;

                              /* FILE 型データの初期設定 */

   for (fp=_iob; fp<_iob+_nfiles; fp++) {
        fp -> _bufptr=NULL ;                    /* バッファポインタのクリア */
        fp -> _bufcnt=0 ;                       /* バッファカウンタのクリア */
        fp -> _buflen=0 ;                       /* バッファ長のクリア*/
        fp -> _bufbase=NULL ;                   /* ベースポインタのクリア*/
        fp -> _ioflag1=0 ;                      /* I/O フラグのクリア*/
        fp -> _ioflag2=0 ;
        fp -> _iofd=0 ;
   }

                              /* 標準入出力ファイルのオープン */

   if (freopen("stdin"*1, "r", stdin)==NULL)      /* 標準入力ファイルのオープン */
        stdin->_ioflag1=0xff ;                    /* ファイルアクセスの禁止 *2 */
   stdin->_ioflag1 |= _IOUNBUF ;                  /* データのバッファリング無 *3 */

   if (freopen("stdout"*1,"w",stdout)==NULL)      /* 標準出力ファイルのオープン */
        stdout->_ioflag1=0xff ;
   stdout->_ioflag1 |= _IOUNBUF ;
   if (freopen("stderr"*1,"w",stderr)==NULL)      /* 標準エラーファイルのオープン*/
        stderr->_ioflag1=0xff ;
   stderr->_ioflag1 |= _IOUNBUF ;
}
```

【注】 *1：**標準入出力ファイルのファイル名を指定します。この名前は、低水準インタフェースルーチン「**open**」で使用します。**

*2：**ファイルのオープンが失敗した場合、ファイルアクセス禁止のフラグを立てます。**

*3：**コンソール等の対話的な装置の場合、バッファリングを行なわないためのフラグを立てます。**

*4：**入出力ファイル数を指定します。指定がない場合は、ファイル数は** 20 **個とみなします。マルチリンケージ処理を使用してロードモジュールを作成するとき、すでにライブラリをリンク済のリロケータブルロードモジュールに対して、上記オブジェクトファイルのリンクを指定すると、二重定義エラーになります。ライブラリをリンクする前に、必ず上記オブジェクトファイルをリンクしてください。**

```
                /* ファイル型データのC言語での宣言 */


    struct _iobuf{
                unsigned char *_bufptr;  /* バッファへのポインタ     */
                long          _bufcnt;   /* バッファカウンタ         */
                unsigned char *_bufbase; /* バッファベースポインタ   */
                long           _buflen;  /* バッファ長               */
                char           _ioflag1; /*  i/oフラグ               */
                char           _ioflag2; /*  i/oフラグ               */
                char           _iofd;    /*  i/oフラグ               */
    }iob[_nfiles];
```

図 3.6　FILE **型データ**

（b）標準入出力以外の初期設定ルーチン（_INIT_OTHERLIB）の作成例

標準入出力以外で初期設定が必要な C ライブラリ関数の初期設定プログラム例を以下に示します。

■例

```
#include <stddef.h>

extern char *_s1ptr ;
extern void srand(unsigned int) ;

void _INIT_OTHERLIB(void)
{

    srand(1) ;              /* rand 関数を使用する場合の初期値の設定 */
    _s1ptr=NULL ;           /* strtok 関数で使用するポインタの初期設定 */
}
```

(5) ファイルのクローズ（__CLOSEALL）

通常ファイルへの出力は、メモリ領域上のバッファにためておき、バッファが一杯になったときに実際の外部記憶装置への書き出しを行ないます。したがってファイルのクローズを行なわないと、ファイルへの出力内容が外部記憶装置へ書き出されないことがあります。

機器組み込み用のプログラムの場合、通常プログラムが終了することはありません。しかし、プログラムの誤りなどにより main 関数が終了する場合、オープンしているファイルは、すべてクローズしなければなりません。

本処理は、main 関数終了時にオープンしているファイルのクローズを行ないます。

ファイルのクローズを行なうプログラム例を以下に示します。

■例

```
#include <stdio.h>

void _CLOSEALL( )      /* アセンブリルーチンのシンボル名から下線を一つ削除 */
{

       int i;

       for (i=0; i<_nfiles; i++)

              /* ファイルがオープンしているかどうかのチェック */

              if(_iob[i]._ioflag1 & ( _IOREAD｜_IOWRITE｜_IORW))

              /* オープンしているファイルのクローズ */

                            fclose(&_iob[i]) ;
}
```

（6）低水準インタフェースルーチン

標準入出力、メモリ管理ライブラリをＣプログラムで使用する場合は、低水準インタフェースルーチンを作成しなければなりません。**表** 3.2 にＣライブラリ関数で使用している低水準インタフェースルーチンの一覧を示します。

表 3.2　**低水準インタフェースルーチンの一覧**

No.	名　　称	機　　能
1	open	**ファイルのオープン**
2	close	**ファイルのクローズ**
3	read	**ファイルからの読み込み**
4	write	**ファイルへの書き出し**
5	lseek	**ファイルの読み込み／書き出しの位置の設定**
6	sbrk	**メモリ領域の確保**

低水準インタフェースルーチンで必要な初期化は、プログラム起動時に行なう必要があります。これは、**「**3.4（4）C **ライブラリ関数の初期設定（**__INITLIB**）」**の中の「_INIT_LOWLEVEL」という関数の中で行なってください。

以下、低水準入出力の基本的な考え方を説明したあと、各インタフェースルーチンの仕様を説明します。また、H8S，H8/300 シリーズシミュレータ・デバッガ上で実行する低水準インタフェースルーチン例を**「付録** E.　**低水準インタフェースルーチンの作成例」**に示しますので、合わせて参照してください。

■注意

関数名 open、close、read、write、lseek、sbrk は低水準インタフェースルーチンの予約語です。ユーザのＣプログラム中では使用しないでください。

（a）入出力の考え方

標準入出力ライブラリでは、ファイルを FILE 型のデータによって管理しますが、低水準インタフェースルーチンでは、実際のファイルと 1 対 1 に対応する正の整数を与え、これによって管理します。この整数をファイル番号といいます。

open ルーチンでは、与えられたファイル名に対してファイル番号を与えます。open ルーチンでは、この番号によってファイルの入出力ができるように、以下の情報を設定する必要があります。

(1) ファイルのデバイスの種類（コンソール、プリンタ、ディスクファイル等）。（コンソールやプリンタ等の特殊なデバイスに対しては、特別なファイル名をシステムで決めておいて open ルーチンで判定する必要があります。）
(2) ファイルのバッファリングをする場合はバッファの位置、サイズ等の情報。
(3) ディスクファイルならば、ファイルの先頭から次に読み込み／書き出しを行なう位置までのバイトオフセット。

open ルーチンで設定した情報に基づいて、以後、入出力（read、write ルーチン）、読み込み／書き出し位置の設定（lseek ルーチン）を行ないます。

close ルーチンでは、出力ファイルのバッファリングを行なっている場合はバッファの内容を実際のファイルに書き出し、open ルーチンで設定したデータの領域が再使用できるようにしてください。

(b) 低水準インタフェースルーチンの仕様

本項では低水準インタフェースルーチンを作成するための仕様を説明します。以下、各ルーチンごとに、ルーチンを呼び出す際のインタフェースとその動作および実現上の注意事項を示します。

各ルーチンのインタフェースは以下の形式で示します。なお、低水準インタフェースルーチンは必ず原型宣言してください。

■凡例

<table>
<tr><td colspan="5">（ルーチン名）</td></tr>
<tr><td>機　　能</td><td colspan="4">（ルーチンの機能概要を示します。）</td></tr>
<tr><td>インタフェース</td><td colspan="4">（ルーチンの C の関数としての宣言方法を示します。）</td></tr>
<tr><td rowspan="3">引　　数</td><td>No.</td><td>名　　前</td><td>型</td><td>意　　味</td></tr>
<tr><td>1</td><td>（インタフェースに示す引数名です。）</td><td>（引数の型を示します。）</td><td>（引数として渡される値の意味を示します。）</td></tr>
<tr><td>・
・
・</td><td>・
・
・</td><td>・
・
・</td><td>・
・
・</td></tr>
<tr><td rowspan="3">リターン値</td><td>型</td><td colspan="3">（リターン値の型を示します。）</td></tr>
<tr><td>正常</td><td colspan="3">（正常に終了した場合のリターン値の意味を示します。）</td></tr>
<tr><td>異常</td><td colspan="3">（エラーが生じた場合のリターン値を示します。）</td></tr>
</table>

(a) open ルーチン				
機　　能	ファイルをオープンします。			
インタフェース	int open (char *name, int mode, int flg);			
引　　数	No.	名　　前	型	意　　味
	1	name	char 型を指すポインタ	ファイルのファイル名を指す文字列
	2	mode	int	ファイルをオープンするときの処理の指定
	3	flg	int	ファイルをオープンするときの処理の指定 （常に 0777）
リターン値	型	int		
	正常	オープンしたファイルのファイル番号		
	異常	−1		

■説明

第 1 引数のファイル名に対応するファイルを操作するための準備をします。open ルーチンでは、後で読み込み／書き出しを行なうために、ファイルの種類（コンソール、プリンタ、ディスクファイル等）を決定しなければなりません。ファイルの種類は、以後 open ルーチンで返したファイル番号を用いて読み込み／書き出しを行なうたびに参照する必要があります。

第 2 引数の mode は、ファイルをオープンする時の処理の指定です。このデータの各ビットの意味について以下に示します。

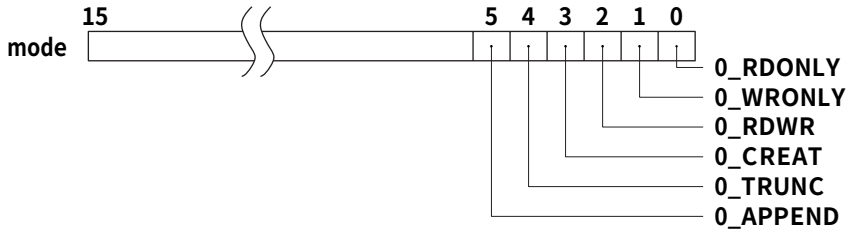


表 3.3　open **ルーチン** mode **ビット説明**

O_RDONLY（0 **ビット**）	**このビットが** 1 **のとき、ファイルを読み込み専用にオープン**
O_WRONLY（1 **ビット**）	**このビットが** 1 **のとき、ファイルを書き出し専用にオープン**
O_RDWR（2 **ビット**）	**このビットが** 1 **のとき、ファイルを読み込み、書き出し両用にオープン**
O_CREAT（3 **ビット**）	**このビットが** 1 **のとき、ファイル名で示すファイルが存在しない場合にファイルを新規に作成**
O_TRUNC（4 **ビット**）	**このビットが** 1 **のとき、ファイル名で示すファイルが存在する場合にファイルの内容を捨て、ファイルのサイズを** 0 **に更新**
O_APPEND（5 **ビット**）	**次に読み込み／書き出しを行なうファイル内の位置を設定** **ビットが** 0 **のとき、：ファイルの先頭に設定** **ビットが** 1 **のとき、：ファイルの最後に設定**

mode で示したファイルの処理の指定と実際のファイルの性質が矛盾する場合はエラーにしてください。

正常にファイルがオープンできた場合は、以後の read、write、lseek、close ルーチンで使用されるファイル番号（正の整数）を返してください。ファイル番号と実際のファイルの対応は低水準インタフェースルーチンで管理する必要があります。オープンに失敗した場合は－1 を返してください。

(b) close ルーチン				
機　能	ファイルをクローズします。			
インタフェース	`int close (int fileno);`			
引　数	No.	名　前	型	意　味
	1	fileno	int	クローズするファイル番号
リターン値	型	int		
	正常	0		
	異常	−1		

■説明

open ルーチンで得られたファイル番号が引数として渡されます。

open ルーチンで設定したファイル管理情報の領域を、再び使用できるように解放してください。また、低水準インタフェースルーチン内で出力ファイルのバッファリングを行なっている場合は、バッファの内容を実際のファイルに書き出してください。

ファイルを正常にクローズできた場合は 0、失敗した場合は−1 を返してください。

<table>
<tr><td colspan="5">(c) read ルーチン</td></tr>
<tr><td>機　　能</td><td colspan="4">ファイルからのデータの読み込みを行ないます。</td></tr>
<tr><td>インタフェース</td><td colspan="4">int read (int fileno,
char *buf,
unsigned int count);</td></tr>
<tr><td rowspan="4">引　　数</td><td>No.</td><td>名　　前</td><td>型</td><td>意　　味</td></tr>
<tr><td>1</td><td>fileno</td><td>int</td><td>読み込みの対象となるファイル番号</td></tr>
<tr><td>2</td><td>buf</td><td>char 型を指すポインタ</td><td>読み込んだデータを設定する領域</td></tr>
<tr><td>3</td><td>count</td><td>unsigned int</td><td>読み込むバイト数</td></tr>
<tr><td rowspan="3">リターン値</td><td>型</td><td colspan="3">int</td></tr>
<tr><td>正常</td><td colspan="3">実際に読み込まれたバイト数</td></tr>
<tr><td>異常</td><td colspan="3">－1</td></tr>
</table>

■説明

第 1 引数（fileno）で示すファイルから、第 2 引数（buf）の指す領域へデータを読み込みます。読み込むデータのバイト数は第 3 引数（count）で示します。

ファイルが終了した場合、count で示されたバイト数以下のバイト数しか読み込むことができません。

ファイルの読み込み／書き出しの位置は、読み込んだバイト数だけ先に進みます。

正常に読み込みができた場合は、実際に読み込んだバイト数を返してください。読み込みに失敗した場合は－1 を返してください。

(d) write ルーチン				
機　　能	ファイルへのデータの書き出しを行ないます。			
インタフェース	int write (int fileno, char *buf, unsigned int count);			
引　　数	No.	名　　前	型	意　　味
	1	fileno	int	書き出しの対象となるファイル番号
	2	buf	char 型を指すポインタ	書き出すデータの領域
	3	count	unsigned int	書き出すバイト数
リターン値	型	int		
	正常	実際に書き出されたバイト数		
	異常	−1		

■説明

第 2 引数（buf）の指す領域から、第 1 引数（fileno）の示すファイルにデータを書き出します。書き込むデータのバイト数は第 3 引数（count）で示します。

ファイルを書き出そうとしているデバイス（ディスク等）が満杯の時は、count で示されたバイト数以下のバイト数しか書き出すことができません。実際に書き出すことのできたバイト数が何度か連続して 0 バイトの場合、ディスクが満杯であると判断してエラー（−1）を返すように実現することをお勧めします。

ファイルの読み込み／書き出しの位置は、書き出したバイト数だけ先に進みます。

正常に書き出しができた場合は、実際に書き出したバイト数を返してください。書き出しに失敗した場合は−1 を返してください。

(e) lseek ルーチン				
機　　能	ファイルの読み込み／書き出しの位置を設定します。			
インタフェース	int lseek (int fileno, long offset, int base);			
引　　数	No.	名　　前	型	意　　味
	1	fileno	int	対象となるファイル番号
	2	offset	long	読み込み／書き出しの位置を示すオフセット（バイト単位）
	3	base	int	オフセットの起点
リターン値	型	long		
	正常	新しいファイルの読み込み／書き出しの位置のファイルの先頭からのオフセット（バイト単位）		
	異常	−1		

■説明

ファイルの読み込み／書き出しを行なうファイル内の位置を、バイト単位で設定します。

新しいファイル内の位置は、第3引数（base）によって、以下の方法で計算し設定してください。

（1）base が 0 のとき

ファイルの先頭から offset バイトの位置に設定します。

（2）base が 1 のとき

現在の位置に offset バイトを加えた位置に設定します。

（3）base が 2 のとき

ファイルのサイズに offset バイトを加えた位置に設定します。

ファイルがコンソールやプリンタ等の対話的なデバイスの場合や、新しいオフセットの値が負になったり、（1）、（2）のときファイルのサイズをこえる場合はエラーにします。

正しくファイル位置を設定できた場合は、新しい読み込み／書き出し位置のファイルの先頭からのオフセットを、そうでない場合は−1 を返してください。

(f) sbrk ルーチン				
機　　能	メモリ領域を割り付けます。			
インタフェース	char *sbrk (int size);			
引　　数	No.	名　　前	型	意　　味
	1	size	int	割り付けるデータのサイズ
リターン値	型	char 型を指すポインタ		
	正常	割り付けた領域の先頭アドレス		
	異常	（char *）－1		

説明

メモリ領域を割り付けるサイズが引数として渡されます。

連続して sbrk ルーチンを呼び出す場合は、下位アドレスから順に連続した領域が割り付けられるようにしてください。

割り付けるメモリ領域が不足した場合はエラーにしてください。

正常に割り付けができた場合は、割り付けた領域の先頭アドレスを、失敗した場合は「（char *）－1」を返してください。

4. C コンパイラのエラーメッセージ

第 4 章　目次

4.1　C コンパイラのエラーメッセージ 139
4.2　C ライブラリ関数のエラーメッセージ..... 160

4.1 C コンパイラのエラーメッセージ

本章では、以下の形式でコンパイラの出力するエラーメッセージとエラー内容を説明します。

エラー番号	**（エラーレベル）エラーメッセージ**
	エラー内容

エラーレベルは、エラーの重要度に従い、５種に分類されます。

表 4.1 C **コンパイラのエラーレベル**

(I)	**インフォメーション**	**処理を継続し、オブジェクトプログラムを出力します。**
(W)	**ウォーニング**	**処理を継続し、オブジェクトプログラムを出力します。**
(E)	**エラー**	**処理を継続します。オブジェクトプログラムは出力しません。**
(F)	**フェータル**	**処理を中断します。**
(-)	**インターナル**	**処理を中断します。**

表 4.2 C **コンパイラのエラーメッセージ**

0001	(I) CHARACTER COMBINATION "/*" IN COMMENT
	注釈の中に、文字列「/*」があります。
0002	(I) NO DECLARATOR
	宣言子のない宣言があります。
0003	(I) UNREACHABLE STATEMENT
	実行されることのない文があります。
0004	(I) CONSTANT AS CONDITION
	if **文または** switch **文の条件を示す式として、定数式を指定しています。**
0005	(I) PRECISION LOST
	代入において、右辺の式の値を左辺の型に変換するときに、精度が失われる可能性があります。
0006	(I) CONVERSION IN ARGUMENT
	関数の引数の式が、原型宣言で指定した引数の型に変換されます。
0008	(I) CONVERSION IN RETURN
	リターン文の式が、関数の返す値の型に変換されます。
0010	(I) ELIMINATION OF NEEDLESS EXPRESSION
	不要な式があります。

0011	(I) USED BEFORE SET SYMBOL : "変数名"
	値の設定されていない局所変数を参照しています。
0015	(I) NO RETURN VALUE
	void 型以外の型を返す関数の中で、リターン文が値を返していないか、またはリターン文がありません。
0100	(I) FUNCTION NOT OPTIMIZED :"関数名"
	関数のサイズが大きすぎるため、最適化できません。
0200	(I) NO PROTOTYPE FUNCTION
	関数の原型宣言がありません。
0300	(I) #PRAGMA INTERRUPT HAS NO EFFECT
	#pragma interrupt で指定された関数が存在しません。
0301	(I) #PRAGMA ABS8 HAS NO EFFECT
	#pragma abs8 で指定された変数が存在しません。
0302	(I) #PRAGMA ABS16 HAS NO EFFECT
	#pragma abs16 で指定された変数が存在しません。
0303	(I) #PRAGMA INDIRECT HAS NO EFFECT
	#pragma indirect で指定された変数が存在しません。
0304	(I) #PRAGMA REGSAVE/NOREGSAVE HAS NO EFFECT
	#pragma regsave/noregsave で指定された変数が存在しません。
0305	(I) #PRAGMA INLINE HAS NO EFFECT
	#pragma inline で指定された変数が存在しません。
0306	(I) #PRAGMA GLOBAL_REGISTER HAS NO EFFECT
	#pragma global_register で指定された変数が存在しません。
1000	(W) ILLEGAL POINTER ASSIGNMENT
	ポインタ型どうしの代入で、それぞれのポインタ型の指す型が異なっています。
1001	(W) ILLEGAL COMPARISON
	二項演算子 == または != の被演算子が、一方がポインタ型で他方が値 0 以外の汎整数型を指しています。
1002	(W) ILLEGAL POINTER REQUIRED
	二項演算子 = =、!=、>、<、>= または <= の被演算子が、同じ型へのポインタ型を指していません。
1005	(W) UNDEFINED ESCAPE SEQUENCE
	文字定数または文字列の中で、文法上定義していない拡張表記（逆スラッシュとそれに続く文字）を用いています。

1007	(W) LONG CHARACTER CONSTANT
	文字定数の長さが 2 文字以上になっています。
1008	(W) IDENTIFIER TOO LONG
	識別子の長さが 31 文字を超えています。
1010	(W) CHARACTER CONSTANT TOO LONG
	文字定数の長さが 2 文字を超えています。
1012	(W) FLOATING POINT CONSTANT OVERFLOW
	浮動小数点定数の値が値の範囲を超えています。符号にしたがって＋∞ または－∞ に対応する内部表現の値を仮定します。
1013	(W) INTEGER CONSTANT OVERFLOW
	整定数の値が unsigned long 型のとり得る値の範囲を超えています。オーバフローした上位ビットを無視した値を仮定します。
1014	(W) ESCAPE SEQUENCE OVERFLOW
	文字定数あるいは文字列の中でのビットパターンを示す拡張表記の値が 255 を超えています。下位 1 バイトの値を有効とします。
1015	(W) FLOATING POINT CONSTANT UNDERFLOW
	浮動小数点定数の値の絶対値が表現できる最小値よりも小さな値となっています。定数の値を 0.0 と仮定します。
1016	(W) ARGUMENT MISMATCH
	原型宣言の中の引数と関数呼び出しの対応する引数の型がポインタ型で、それぞれの指す型が異なっています。関数呼び出しの引数のポインタの内部表現をそのまま設定します。
1017	(W) RETURN TYPE MISMATCH
	関数の返す型とリターン文の式の型がポインタ型で、それぞれの指す型が異なっています。リターン文の式のポインタの内部表現をそのまま設定します。
1018	(W) TYPE MISMATCH
	extern 記憶クラスを持つ同じ名前の変数あるいは関数を、異なる有効範囲で宣言していますが、型が一致していません。
1019	(W) ILLEGAL CONSTANT EXPRESSION
	定数式において関係演算子 <、>、<= または >= の被演算子が、同じ型へのポインタ型を指していません。結果の値を 0 と仮定します。
1020	(W) ILLEGAL CONSTANT EXPRESSION
	定数式において二項演算子 - の被演算子が、同じ型へのポインタ型を指していません。結果の値を 0 と仮定します。

1021	(W) CONVERT TO SJIS-SPACE
	日本語コードで指定の出力コードに変換できないものがあります。シフト JIS のスペースに変換します。
1022	(W) CONVERT TO EUC-SPACE
	日本語コードで指定の出力コードに変換できないものがあります。EUC のスペースに変換します。
1200	(W) DIVISION BY FLOATING POINT ZERO
	定数式の中で浮動小数点数 0.0 を除数とする割り算を行なっています。符号にしたがって、＋∞または－∞に対応する内部表現の値を仮定します。
1201	(W) INEFFECTIVE FLOATING POINT OPERATION
	定数式の中で ∞－∞、0.0/0.0 等の無効演算を行なっています。無効演算の結果を表わす非数に対応する内部表現の値を仮定します。
1300	(W) COMMAND PARAMETER SPECIFIED TWICE
	同じコンパイラオプションを 2 度以上指定しています。同じコンパイラオプションの中で最後に指定したものを有効とします。
1302	(W) 'FRAME' OR 'NOFRAME' OPTION IGNORED
	最適化指定ありのときに frame オプション、または最適化指定なしのときに noframe オプションを指定しています。オプションの指定を無視します。
1303	(W) COMPLETED FILE NAME TOO LONG
	ルートディレクトリを起点としたパス名を含むファイル名の長さが 251 文字を超えています。コマンドライン上で指定したファイル名を debug 情報に出力します。
1305	(W) 'SHOW=OBJECT' OPTION IGNORED
	アセンブリソースプログラムの出力指定時に、show=object オプションを指定しています。オプションの指定を無視します。
1306	(W) 'SPEED=INLINE' OPTION IGNORED
	最適化指定なしのときに speed=inline オプションを指定しています。オプションの指定を無視します。
1307	(W) SECTION NAME TOO LONG
	セクション名称の長さが 32 文字を超えています。32 文字までを有効とし、それ以降の文字を無視します。
1308	(W) 'SPEED=LOOP' OPTION IGNORED
	最適化指定なしのときに speed=loop オプションを指定しています。オプションの指定を無視します。

1310	(W) 'GOPTIMIZE' OPTION IGNORED
	アセンブリソースプログラムの出力指定時に goptimize オプションを指定しています。オプションの指定を無視します。
1311	(W) 'CMNCODE' OPTION IGNORED
	最適化指定なしのときに cmncode オプションを指定しています。オプションの指定を無視します。
1400	(W) #PRAGMA INLINE IS NOT EXPANDED
	#pragma inline で指定した関数がインライン展開されませんでした。#pragma inline 指定を無視します。
1401	(W) #PRAGMA ABS16 IGNORED
	CPU/動作モードが 2600n、2000n、300hn および 300 のときに、#pragma abs16 を指定しています。#pragma abs16 指定を無視します。
1403	(W) #PRAGMA ASM IGNORED
	オブジェクト形式がリロケータブルオブジェクトプログラムのときに、 #pragma asm を指定しています。#pragma asm 指定を無視します。
1404	(W) 'CASE=TABLE' OPTION IGNORED BY SWITCH
	switch 文をテーブル方式に展開することができません。switch 文を if-then 方式で展開します。switch 文をテーブル方式に展開するためには、switch 文の case ラベル値を以下の範囲内にしてください。 （case ラベル値の範囲）＜（case ラベル数×3） case ラベル値の範囲：（case ラベル値の最大値 - case ラベル値の最小値＋1）
1405	(W) ILLEGAL #PRAGMA SYNTAX
	認識できない#pragma 文を指定しています。#pragma 指定を無視します。
2000	(E) ILLEGAL PREPROCESSOR KEYWORD
	プリプロセッサ文で、誤ったキーワードを使用しています。
2001	(E) ILLEGAL PREPROCESSOR SYNTAX
	プリプロセッサ文またはマクロ呼び出しの指定方法に誤りがあります。
2002	(E) "," NOT FOUND
	引数のある#define 文で引数の並びを区切るコンマ（,）が抜けています。
2003	(E) ")" NOT FOUND
	名前が#define 文で定義されているかどうかを判定する defined 式で名前の次の右括弧「)」が抜けています。
2004	(E) ">" NOT FOUND
	#include 文のファイル名の指定でファイル名の次の > がありません。
2005	(E) CANNOT OPEN INCLUDE
	#include 文で指定したファイル名のファイルがオープンできません。

2006	(E) MULTIPLE #DEFINE'S
	#define 文で同じマクロ名を再定義しています。
2008	(E) #ELIF MISMATCHES
	#elif 文に対応する#if 文、#ifdef 文、#ifndef 文、#elif 文がありません。
2009	(E) #ELSE MISMATCHES
	#else 文に対応する#if 文、#ifdef 文、#ifndef 文がありません。
2010	(E) MACRO PARAMETERS MISMATCH
	マクロ呼び出しの引数の数がマクロ定義の引数の数と異なっています。
2011	(E) LINE TOO LONG
	マクロ展開後のソースプログラムの行が 4096 文字を超えています。
2012	(E) KEYWORD AS A MACRO NAME
	プリプロセッサで規定しているキーワードを#define 文または、#undef 文のマクロ名として定義しています。
2013	(E) #ENDIF MISMATCHES
	#endif 文に対応する#if 文、#ifdef 文、#ifndef 文がありません。
2014	(E) #ENDIF EXPECTED
	#if 文、#ifdef 文、#ifndef 文に対応する#endif 文がないままでファイルが終了しました。
2016	(E) PREPROCESSOR CONSTANT EXPRESSION TOO COMPLEX
	#if 文、#elif 文で指定した定数式の演算子と被演算子の合計が 512 個を超えています。
2017	(E) MISSING "
	#include 文のファイル名の指定で、ファイル名の次に " がありません。
2018	(E) ILLEGAL #LINE
	#line 文で指定した行数が 32767 行を超えています。
2019	(E) FILE NAME TOO LONG
	ファイル名の長さが 128 文字を超えています。
2020	(E) SYSTEM IDENTIFIER REDEFINED : "名前"
	組み込み関数と同名のシンボルを定義しています。
2021	(E) SYSTEM IDENTIFIER MISMATCH : "名前"
	指定された CPU/動作モードに存在しない組み込み関数を使用しています。
2100	(E) MULTIPLE STRAGE CLASSES
	宣言の中で二つ以上の記憶クラス指定子を指定しています。
2101	(E) ADDRESS OF REGISTER
	レジスタ記憶クラスを持つ変数に対して、単項演算子&を適用しています。

2102	(E) ILLEGAL TYPE COMBINATION
	型指定子の組み合わせが誤っています。
2103	(E) BAD SELF REFERENCE STRUCTURE
	構造体、共用体のメンバの型を、親の構造体または共用体と同じ型で宣言しています。
2104	(E) ILLEGAL BIT FILED WIDTH
	ビットフィールド幅を示す定数式が整数型でありません。あるいはビットフィールド幅として負の整数を指定しています。
2105	(E) INCOMPLETE TAG USED IN DECLARATION
	構造体または共用体で仮宣言されたタグ名または、未宣言のタグ名を typedef 宣言、ポインタを指す型あるいは関数の返す型以外の宣言で使用しています。
2106	(E) EXTERN VARIABLE INITIALIZED
	複文内で extern 記憶クラスを指定した宣言に対して初期値を指定しています。
2107	(E) ARRAY OF FUNCTION
	要素の型が関数型となる配列型を指定しています。
2108	(E) FUNCTION RETURNING ARRAY
	リターン値の型が配列型となる関数型を指定しています。
2109	(E) ILLEGAL FUNCTION DECLARATION
	複文内の関数型の変数の宣言において、extern 以外の記憶クラスを指定しています。
2110	(E) ILLEGAL STORAGE CLASS
	外部定義の中で記憶クラスとして auto または register を指定しています。
2111	(E) FUNCTION AS A MEMBER
	構造体または共用体のメンバの型に関数型を指定しています。
2112	(E) ILLEGAL BIT FIELD TYPE
	ビットフィールドに誤った型を指定しています。ビットフィールドに許される型指定子は、char, unsigned char, short, unsigned short, int, unsigned int のいずれか、これらの型に const または volatile を組み合わせた型です。
2113	(E) BIT FIELD TOO WIDE
	ビットフィールド幅が型指定子で指定したサイズ（8 ビット、16 ビット）を超えています。
2114	(E) MULTIPLE VARIABLE DECLARATIONS
	変数名を同じ有効範囲の中で重複して宣言しています。

2115	(E) MULTIPLE TAG DECLARATIONS
	構造体、共用体、列挙型のタグ名を同じ有効範囲の中で重複して宣言しています。
2117	(E) EMPTY SOURCE PROGRAM
	ソースプログラム内に外部定義が含まれていません。
2118	(E) PROTOTYPE MISMATCH : "関数名"
	関数の型が以前になされている宣言で指定した型と一致しません。
2119	(E) NOT A PARAMETER NAME : "引数名"
	関数の引数宣言列にない識別子に対して引数宣言を行なっています。
2120	(E) ILLEGAL PARAMETER STRAGE CLASS
	関数の引数宣言で register 以外の記憶クラスを指定しています。
2121	(E) ILLEGAL TAG NAME
	構造体、共用体または列挙型とタグ名の組み合わせが、以前に宣言した型とタグ名の組み合わせと異なっています。
2122	(E) BIT FIELD WIDTH 0
	メンバ名を指定しているビットフィールドの幅が 0 になっています。
2123	(E) UNDEFINED TAG NAME
	列挙型の宣言で未定義のタグ名を使用しています。
2124	(E) ILLEGAL ENUM VALUE
	列挙型のメンバに整数でない定数式を指定しています。
2125	(E) FUNCTION RETURNING FUNCTION
	リターン値の型が関数型となる関数型を指定しています。
2126	(E) ILLEGAL ARRAY SIZE
	配列の要素数を指定する値が制限値を超えています。配列要素数の制限値は、CPU/動作モードが 2600n,2000n,300hn,300 のとき 65535、 2600a:20,2000a:20,300ha:20 のとき 1048575、 2600a:24,2000a:24,300ha:24 のとき 16777215、 2600a:28,2000a:28 のとき 268435455、 2600a:32,2000a:32 のとき 4294967295 です。
2127	(E) MISSING ARRAY SIZE
	配列の要素数の指定がありません。
2128	(E) ILLEGAL POINTER DECLARATION
	ポインタ型の宣言を示す * の直後に const、volatile 以外の型指定子を指定しています。

2129	(E) ILLEGAL INITIALIZER TYPE
	変数の初期値指定において初期値の型が変数に代入可能な型でありません。
2130	(E) INITIALIZER SHOULD BE CONSTANT
	構造体型、共用体型、配列型の変数の初期値、または静的に割り付けられる変数の初期値に定数式でないものを指定しています。
2131	(E) NO TYPE NOR STRAGE CLASS
	外部データ定義において記憶クラスまたは型の指定がありません。
2132	(E) NO PARAMETER NAME
	関数の引数宣言列が空であるにもかかわらず引数の宣言を行なっています。
2133	(E) MULTIPLE PARAMETER DECLARATIONS
	（マクロ）関数定義の引数宣言列の中で同一名の引数を重複して宣言しているか、または引数宣言が関数宣言子の中と外の 2 箇所で行なわれています。
2134	(E) INITIALIZER FOR PARAMETER
	引数の宣言において初期値を指定しています。
2135	(E) MULTIPLE INITIALIZATION
	同一の変数に対して、初期化を重複して行なっています。
2136	(E) TYPE MISMATCH
	extern あるいは static 記憶クラスを持つ変数あるいは関数を 2 度以上宣言しており、その型が一致していません。
2137	(E) NULL DECLARATION FOR PARAMETER
	関数の引数宣言で識別子を指定していません。
2138	(E) TOO MANY INITIALIZERS
	構造体、共用体または配列の初期値指定において、構造体のメンバ数または配列の要素数より多く初期値の数を指定しています。あるいは、共用体の最初のメンバがスカラ型のときに 2 個以上の初期値を指定しています。
2139	(E) NO PARAMETER TYPE
	関数宣言の引数宣言に型指定がありません。
2140	(E) ILLEGAL BIT FIELD
	共用体にビットフィールドを指定しています。
2141	(E) ILLEGAL BIT FIELD
	構造体の先頭のメンバに無名のビットフィールドを指定しています。

2142	(E) ILLEGAL VOID TYPE
	void 型の指定方法に誤りがあります。void 型を指定できるのは以下の三つの場合です。 (1) ポインタの指す先の型として指定する場合。 (2) 関数の返す型として指定する場合。 (3) 原型宣言の関数が引数を持たないことを明示的に指定する場合。
2143	(E) ILLEGAL STATIC FUNCTION
	ソースファイル内に定義のない static 記憶クラスを持つ関数宣言があります。
2200	(E) INDEX NOT INTEGER
	配列の添字の式が整数型ではありません。
2201	(E) CANNOT CONVERT PARAMETER : "n"
	関数呼び出しにおける n 番目の引数を対応する原型宣言の引数の型に変換できません。
2202	(E) NUMBER OF PARAMETERS MISMATCH
	関数呼び出しにおける引数の数が原型宣言の引数の数と一致しません。
2203	(E) ILLEGAL MEMBER REFERENCE
	演算子 . の左側の式の型が構造体型、共用体型ではありません。
2204	(E) ILLEGAL MEMBER REFERENCE
	演算子 -> の左側の式の型が構造体型または共用体型へのポインタではありません。
2205	(E) UNDEFINED MEMBER NAME
	構造体、共用体への参照で宣言していないメンバ名を使用しています。
2206	(E) MODIFIABLE LVALUE REQUIRED
	前置または後置演算子 ++、-- を代入可能な左辺値(配列型、const 型を除く左辺値)でない式に使用しています。
2207	(E) SCALAR REQUIRED
	単項演算子 ! をスカラ型でない式に使用しています。
2208	(E) POINTER REQUIRED
	単項演算子 * をポインタ型でない式か、または void 型へのポインタ型の式に使用しています。
2209	(E) ARITHMETIC TYPE REQUIRED
	単項演算子 + または - を算術型でない式に使用しています。
2210	(E) INTEGER REQUIRED
	単項演算子 ~ を汎整数型でない式に使用しています。

2211	(E) ILLEGAL SIZEOF
	sizeof 演算子をビットフィールドの指定のあるメンバ、関数型、void 型またはサイズの指定していない配列に使用しています。
2212	(E) ILLEGAL CAST
	キャスト演算子で指定している型が配列型、構造体型または共用体型です。あるいはキャスト演算子の被演算子が void 型、構造体型か共用体型で型変換できません。
2213	(E) ARITHMETIC TYPE REQUIRED
	二項演算子 *、/、*= または /= を算術型でない式に適用しています。
2214	(E) INTEGER REQUIRED
	二項演算子 <<、>>、&、|、^、%、<<=、>>=、&=、|=、^= または %= を汎整数型でない式に適用しています。
2215	(E) ILLEGAL TYPE FOR +
	二項演算子 + の被演算子の型の組み合わせが許されていません。二項演算子 + の型の組み合わせで許されるのは、両辺とも算術型の場合と、一方がポインタ型で他方が汎整数型の場合だけです。
2216	(E) ILLEGAL TYPE REQUIRED FOR PARAMETER
	関数呼び出しの引数の型に void 型を指定しています。
2217	(E) ILLEGAL TYPE FOR -
	二項演算子 - の被演算子の型の組み合わせが許されていません。二項演算子 - の型の組み合わせで許されるのは、以下の三つの場合です。 (1) 両方の被演算子が算術型の場合。 (2) 両方の被演算子が同じ型へのポインタ型の場合。 (3) 第１被演算子がポインタ型で、第 2 被演算子が汎整数型の場合。
2218	(E) SCALAR REQUIRED
	条件演算子 ? : の第 1 被演算子の型がスカラ型でありません。
2219	(E) TYPE NOT COMPATIBLE
	条件演算子 ? : の第 2 被演算子と第 3 被演算子の型が合っていません。条件演算子 ? : の第 2 被演算子と第 2 被演算子の組み合わせで許されるのは、以下の六つの場合です。 (1) 両方とも算術型の場合。 (2) 両方とも void 型の場合。 (3) 両方とも同じ型へのポインタ型の場合。 (4) 一方がポインタ型で、もう一方が値 0 の整定数または値 0 の整定数を void 型へのポインタ型に変換したものである場合。 (5) 一方がポインタ型で、もう一方が void 型へのポインタ型の場合。 (6) 両方とも同じ型の構造体または共用体の場合。

2220	(E) MODIFIEBLE LVALUE REQUIRED
	代入演算子 =、*=、/=、%=、+=、-=、<<=、>>=、&=、^= または |= の左辺の式に代入可能な左辺値(配列型、const 型を除く左辺値)以外の式を指定しています。
2221	(E) ILLEGAL TYPE FOR POSTINCREMENT OR POSTDECREMENT
	後置演算子 ++ または -- の被演算子にスカラ型以外の型、関数型または void 型へのポインタ型を指定しています。
2222	(E) TYPE NOT COMPATIBLE
	代入演算子 = の両辺の式の型が合っていません。代入演算子 = の両辺の式の組み合わせで許されるのは、以下の五つの場合です。 (1) 両方とも算術型の場合。 (2) 両方とも同じ型へのポインタ型の場合。 (3) 左辺がポインタ型で、右辺が値 0 の整定数または値 0 の整定数を void 型へのポインタ型に変換したものである場合。 (4) 一方がポインタ型で、もう一方が void 型へのポインタ型の場合。 (5) 両方とも同じ型の構造体または共用体の場合。
2223	(E) INCOMPLETE TAG USED IN EXPRESSION
	構造体または共用体で仮宣言されたタグ名を式中で使用しています。
2224	(E) ILLEGAL TYPE FOR ASSIGN
	代入演算子 += または -= の両辺の型が正しくありません。
2225	(E) UNDECLARED NAME : "名前"
	宣言していない名前を式の中で用いています。
2226	(E) SCALAR REQUIRED
	二項演算子 && または || をスカラ型でない式に適用しています。
2227	(E) ILLEGAL TYPE FOR EQUALITY
	等値演算子 == または != の被演算子の型の組み合わせが許されていません。 等値演算の被演算子の組み合わせで許されるのは、以下の三つの場合です。 (1) 両方とも算術型の場合。 (2) 両方とも同じ型へのポインタ型の場合。 (3) 一方がポインタ型でもう一方が値 0 の整定数または void 型へのポインタ型である場合。

2228	(E) ILLEGAL TYPE FOR COMPARISON
	関係演算子 >、<、>= または <= の被演算子の型の組み合わせが許されていません。関係演算子の被演算子の組み合わせで許されるのは、以下の二つの場合です。 (1) 両方とも算術型の場合。 (2) 両方とも同じ型へのポインタ型の場合。
2230	(E) ILLEGAL FUNCTION CALL
	関数呼び出しにおいて、関数型あるいは関数型へのポインタ型でない式を用いています。
2231	(E) ADDRESS OF BIT FIELD
	単項演算子 & をビットフィールドに適用しています。
2232	(E) ILLEGAL TYPE FOR PREINCREMENT OR PREDECREMENT
	前置演算子 ++ または -- の被演算子にスカラ型以外の型、関数型または void 型へのポインタ型を指定しています。
2233	(E) ILLEGAL ARRAY REFERENCE
	配列型、関数型または void 型を除くポインタ型以外の式を配列として使用しています。
2234	(E) ILLEGAL TYPEDEF NAME REFERENCE
	typedef 宣言された名前を式の中で変数として使用しています。
2235	(E) ILLEGAL CAST
	ポインタを浮動小数点型にキャストしています。
2236	(E) ILLEGAL CAST IN CONSTANT
	CPU/動作モードが 300,300hn,2600n,2000n の場合、定数式でポインタ型を char 型または long 型にキャストしています。 CPU/動作モードが 300ha,2600a,2000a の場合、定数式でポインタ型を char 型、short 型または int 型にキャストしています。
2237	(E) ILLEGAL CONSTANT EXPRESSION
	定数式の中でポインタ型の定数を整数型へキャストした結果に対して演算を行なっています。
2238	(E) LVALUE OR FUNCTION TYPE REQUIRED
	単項演算子＆を左辺値あるいは関数型以外の式に適用しています。
2300	(E) CASE NOT IN SWITCH
	case ラベルを switch 文以外に指定しています。
2301	(E) DEFAULT NOT IN SWITCH
	default ラベルを switch 文以外に指定しています。

2302	(E) MULTIPLE LABELS
	一つの関数内にラベル名を重複して定義しています。
2303	(E) ILLEGAL CONTINUE
	continue 文を while 文、for 文または do 文以外に指定しています。
2304	(E) ILLEGAL BREAK
	break 文を while 文、for 文、do 文または switch 文以外に指定しています。
2305	(E) VOID FUNCTION RETURNS VALUE
	void 型を返す関数の中の return 文でリターン値を指定しています。
2306	(E) CASE LABEL NOT CONSTANT
	case ラベルの式が汎整数型の定数式ではありません。
2307	(E)MULTIPLE CASE LABELS
	同一の値を持つ case ラベルを一つの switch 文の中に重複して指定しています。
2308	(E) MULTIPLE DEFAULT LABELS
	default ラベルを一つの switch 文の中に重複して指定しています。
2309	(E) NO LABEL FOR GOTO
	goto 文で指定した行き先のラベルがありません。
2310	(E) SCALAR REQUIRED
	while 文、for 文または do 文の制御式（文の実行を判定する式）がスカラ型ではありません。
2311	(E) INTEGER REQUIRED
	switch 文の制御式（文の実行を判定する式）が汎整数型ではありません。
2312	(E) MISSING (
	if 文、while 文、for 文、do 文または switch 文の制御式（文の実行を判定する式）の左括弧「 (」がありません。
2313	(E) MISSING ;
	do 文の最後のセミコロン(;)がありません。
2314	(E) SCALAR REQUIRED
	if 文の制御式（文の実行を判定する式）がスカラ型ではありません。
2316	(E) ILLEGAL TYPE FOR RETURN VALUE
	return 文の式の型を関数の返す型に変換することができません。
2320	(E) ILLEGAL ASM POSITION
	#pragma asm の指定位置が適切ではありません。
2330	(E) ILLEGAL INTERRUPT FUNCTION DECLARATION
	割り込み関数宣言に誤りがあります。

2331	(E) ILLEGAL INTERRUPT FUNCTION CALL
	割り込み関数宣言のある関数をプログラム中で呼び出しまたは参照しています。
2332	(E) INTERRUPT FUNCTION ALREADY DECLARED
	割り込み関数宣言#pragma interrupt で指定した関数を既に通常の関数として宣言しています。
2333	(E) MULTIPLE INTERRUPT FOR ONE FUNCTION
	同一関数に対して割り込み関数宣言#pragma interrupt を重複して宣言しています。
2334	(E) ILLEGAL PARAMETER IN INTERRUPT FUNCTION
	割り込み関数で使用する引数の型が一致していません。引数の型として指定できるのは void 型だけです。
2335	(E) MISSING PARAMETER DECLARATION IN INTERRUPT FUNCTION
	割り込み関数宣言#pragma interrupt のスタック切り替え指定 (sp) または割り込み終了の指定 (sy) に、宣言していない変数または関数を使用しています。
2336	(E) PARAMETER OUT OF RANGE IN INTERRUPT FUNCTION
	割り込み関数宣言#pragma interrupt の引数 tn の値が 3 を超えています。
2337	(E) ILLEGAL INTERRUPT TYPE
	割り込み関数宣言に誤りがあります。
2340	(E) ILLEGAL ABS8 DECLARATION
	短絶対アドレス変数宣言に誤りがあります。
2341	(E) ABS8 ALREADY DECLARED
	短絶対アドレス変数宣言#pragma abs8 で指定した変数を既に変数として宣言しています。
2342	(E) ILLEGAL ABS8 TYPE
	短絶対アドレス変数宣言#pragma abs8 で指定した変数を変数名以外の型で宣言しています。
2345	(E) ILLEGAL ABS16 DECLARATION
	短絶対アドレス変数宣言に誤りがあります。
2346	(E) ABS16 ALREADY DECLARED
	短絶対アドレス変数宣言#pragma abs16 で指定した変数を既に変数として宣言しています。

2347	(E) ILLEGAL ABS16 TYPE
	短絶対アドレス変数宣言#pragma abs16 で指定した変数を変数名以外の型で宣言しています。
2350	(E) ILLEGAL SECTION NAME DECLARATION
	#pragma section の指定に誤りがあります。
2352	(E) SECTION NAME TABLE OVERFLOW
	通常のセクション、8 ビット／16 ビット絶対アドレスセクションおよび、メモリ間接呼び出しセクションの各セクション数が 64 個を超えたか、または全体で 256 個を超えました。
2360	(E) ILLEGAL INDIRECT FUNCTION DECLARATION
	メモリ間接関数宣言に誤りがあります。
2361	(E) INDIRECT FUNCTION ALREADY DECLARED
	メモリ間接関数宣言#pragma indirect で指定した関数を既に関数として宣言しています。
2362	(E) ILLEGAL INDIRECT TYPE
	メモリ間接関数宣言#pragma indirect で指定した関数を関数以外の型で宣言または定義しています。
2363	(E) TOO MANY INDIRECT IDENTIFIERS
	1 ファイルで指定できるメモリ間接関数の数が制限を超えました。1 ファイルで指定できるメモリ間接関数の数は、256 個です。
2370	(E) ILLEGAL REGSAVE/NOREGSAVE DECLARATION
	#pragma regsave または#pragma noregsave 宣言に誤りがあります。
2371	(E) REGSAVE/NOREGSAVE FUNCTION ALREADY DECLARED
	#pragma regsave または#pragma noregsave で指定した関数を既に関数として宣言しています。
2372	(E) ILLEGAL REGSAVE/NOREGSAVE TYPE
	#pragma regsave または#pragma noregsave で指定した関数を関数以外の型で宣言または定義しています。
2380	(E) ILLEGAL INLINE/INLINE_ASM DECLARATION
	#pragma inline または#pragma inline_asm 宣言に誤りがあります。
2381	(E) INLINE/INLINE_ASM FUNCTION ALREADY DECLARED
	#pragma inline または#pragma inline_asm で指定した関数を既に関数として宣言しています。

2382	(E) ILLEGAL INLINE/INLINE_ASM TYPE
	#pragma inline **または**#pragma inline_asm **で指定した関数を関数以外の型で宣言または定義しています。**
2383	(E) #PRAGMA INLINE_ASM IGNORED
	オブジェクト形式がリロケータブルオブジェクトプログラムのときに、#pragma inline_asm **を指定しています。**
2390	(E) ILLEGAL GLOBAL_REGISTER DECLARATION
	#pragma global_register **宣言に誤りがあります。**
2391	(E) GLOBAL_REGISTER ALREADY DECLARED
	#pragma global_register **で指定した変数を既に変数として宣言しています。**
2392	(E) ILLEGAL GLOBAL_REGISTER TYPE
	#pragma global_register **で指定した変数を変数以外の型で宣言しています。**
2393	(E) ILLEGAL REGISTER
	#pragma global_register **で指定したレジスタ名に誤りがあります。または、重複指定しています。**
2400	(E)ILLEGAL CHARACTER : "**文字**"
	不正な印字文字があります。
2401	(E) INCOMPLETE CHARACTER CONSTANT
	文字定数の途中に改行があります。
2402	(E) INCOMPLETE STRING
	文字列の途中に改行があります。
2403	(E) EOF IN COMMENT
	コメントの途中でファイルが終了しました。
2404	(E) ILLEGAL CHARACTER CODE :"**文字コード**"
	不正な文字コードがあります。
2405	(E) NULL CHARACTER CONSTANT
	文字定数の中に文字を指定していません。すなわち '' という形式の文字定数を指定しています。
2406	(E) OUT OF FLOAT
	浮動小数点定数の有効桁数が 17 **桁を超えています。**
2407	(E) INCOMPLETE LOGICAL LINE
	空でないソースファイルの最後の文字に、バックスラッシュ（＼）またはバックスラッシュのあとに改行文字（＼（RET**））を指定しています。**
2408	(E) COMMENT NEST TOO DEEP
	コメントのネストが 255 **レベルを超えています。**

2500	(E) ILLEGAL TOKEN : "語句"
	語句の並びが文法に合っていません。
2501	(E) DIVISION BY ZERO
	定数式中で整数型データのゼロ除算が行なわれました。
2600	(E) #ERROR DIAGNOSTIC MESSAGE : "文字列"
	nolist オプションが指定されていなければ、#error の文字列で指定されたエラーメッセージをリストファイルに表示します。
2801	(E) ILLEGAL PARAMETER TYPE IN INLINE FUNCTION
	組み込み関数で引数の型が一致しません。
2802	(E) PARAMETER OUT OF RANGE IN INLINE FUNCTION
	組み込み関数で引数の大きさが指定可能範囲を超えています。
3000	(F) STATEMENT NEST TOO DEEP
	if 文、while 文、for 文、do 文および switch 文のネストが、32 レベルを超えています。
3001	(F) BLOCK NEST TOO DEEP
	複文のネストが 32 レベルを超えています。
3002	(F) #IF NEST TOO DEEP
	条件コンパイル（#if、#ifdef、#ifndef、#elif、#else）のネストが 32 レベルを超えています。
3006	(F) TOO MANY PARAMETERS
	関数の宣言または呼び出しにおいて引数の数が 63 個を超えています。
3007	(F) TOO MANY MACRO PARAMETERS
	マクロの定義または呼び出しにおいて、引数の数が 63 個を超えています。
3008	(F) LINE TOO LONG
	マクロ展開後の 1 行の長さが 4096 文字を超えています。
3009	(F) STRING LITERAL TOO LONG
	文字列の長さが 512 文字を超えています。文字列の長さは、連続して指定した文字列を連結した後のバイト数です。ここでいう文字列の長さとは、ソースプログラム上の長さではなく文字列のデータに含まれるバイト数で、拡張表記も 1 文字に数えます。
3010	(F) #INCLUDE NEST TOO DEEP
	#include 文によるファイルの取り込みのネストが 30 レベルを超えています。
3011	(F) MACRO EXPANSION NEST TOO DEEP
	#define 文によるマクロ名の再置換が 32 個を超えています。

3012	(F) TOO MANY FUNCTION DEFINITIONS
	関数定義の数が 512 個を超えています。
3013	(F) TOO MANY SWITCHES
	switch 文の数が 256 個を超えています。
3014	(F) FOR NEST TOO DEEP
	for 文のネストが 16 レベルを超えています。
3015	(F) SYMBOL TABLE OVERFLOW
	コンパイラが生成するシンボルの数が 24576 を超えています。
3016	(F) INTERNAL LABEL OVERFLOW
	コンパイラが生成する内部ラベルの数が 49152 を超えています。
3017	(F) TOO MANY CASE LABELS
	一つの switch 文の中の case ラベルの数が 511 個を超えています。
3018	(F) TOO MANY GOTO LABELS
	一つの関数の中で定義している goto ラベルの数が 511 個を超えています。
3019	(F) CANNOT OPEN SOURCE FILE
	ソースファイルをオープンすることができません。
3020	(F) SOURCE FILE INPUT ERROR
	ソースファイルまたはインクルードファイルを読み込むことができません。
3021	(F) MEMORY OVERFLOW
	コンパイラが内部で使用するメモリ領域を割り当てることができません。
3022	(F) SWITCH NEST TOO DEEP
	switch 文のネストが 16 レベルを超えています。
3023	(F) TYPE NEST TOO DEEP
	基本型を修飾する型（ポインタ型、配列型、関数型）の数が 16 個を超えています。
3024	(F) ARRAY DIMENSION TOO DEEP
	配列の次元数が 6 次元を超えています。
3025	(F) SOURCE FILE NOT FOUND
	コマンドラインの中にソースファイル名の指定がありません。
3026	(F) EXPRESSION TOO COMPLEX
	式が複雑すぎます。
3027	(F) SOURCE FILE TOO COMPLEX
	プログラムの文のネストが深いかあるいは、式が複雑すぎます。
3028	(F) SOURCE LINE NUMBER OVERFLOW
	ソース行番号が 65535 行を超えています。

3031	(F) DATA SIZE OVERFLOW
	配列または構造体の大きさが、制限値を超えています。配列または構造体の制限値は、CPU/動作モードが 2600n,2000n,300hn,300 のとき 65535、 2600a:20,2000a:20,300ha:20 のとき 1048575、 2600a:24,2000a:24,300ha:24 のとき 16777215、 2600a:28,2000a:28 のとき 268435455、 2600a:32,2000a:32 のとき 4294967295 です。
3033	(F) SYMBOL TABLE OVERFLOW
	デバッグ情報のためのシンボルの数が、65535 個を超えています。
3200	(F) OBJECT SIZE OVERFLOW
	オブジェクトサイズがメモリの制限を超えています。オブジェクトサイズの制限値は、 CPU/動作モードが 2600n,2000n,300hn,300 のとき 65535、 2600a:20,2000a:20,300ha:20 のとき 1048575、 2600a:24,2000a:24,300ha:24 のとき 16777215、 2600a:28,2000a:28 のとき 268435455、 2600a:32,2000a:32 のとき 4294967295 です。
3202	(F) ILLEGAL STACK ACCESS
	局所変数・テンポラリ領域およびレジスタ退避領域がスタックポインタ(SP)またはフレームポインタ(FP)から制限値より離れています。または、引数領域が SP または FP から制限値より離れています。SP,FP からのオフセットの制限値は、CPU/動作モードが 2600n,2000n,300hn,300 のとき 32767、 2600a:20,2000a:20,300ha:20 のとき 524287、 2600a:24,2000a:24,300ha:24 のとき 8388607、 2600a:28,2000a:28 のとき 134217727、 2600a:32,2000a:32 のとき 2147483647 です。
3204	(F)TOO MANY SOURCE LINES FOR DEBUG
	デバッグのための実行文の数が制限値を超えています。
3300	(F) CANNOT OPEN INTERNAL FILE
	コンパイラが内部で使用する中間ファイルをオープンすることができません。
3301	(F) CANNOT CLOSE INTERNAL FILE
	コンパイラが内部で生成する中間ファイルをクローズすることができません。コンパイラが生成する中間ファイルをアクセスしていないかを確認してください。

3302	(F) CANNOT INPUT INTERNAL FILE
	コンパイラが内部で生成する中間ファイルを読み込むことができません。コンパイラが生成する中間ファイルをアクセスしていないかを確認してください。
3303	(F) CANNOT OUTPUT INTERNAL FILE
	コンパイラが内部で生成する中間ファイルに書き込むことができません。ディスク容量を増やしてください。
3304	(F) CANNOT DELETE INTERNAL FILE
	コンパイラが内部で生成する中間ファイルを削除することができません。コンパイラが生成する中間ファイルをアクセスしていないかを確認してください。
3305	(F) INVALID COMMAND PARAMETER "オプション名"
	コンパイラオプションの指定方法が誤っています。
3306	(F) INTERRUPT IN COMPILATION
	コンパイル処理中に標準入力端末から (CNTL) C コマンドによる割り込みを検出しました。
3307	(F) COMPILER VERSION MISMATCH "ファイル名"
	コンパイラを構成するファイル間のバージョンが一致していません。インストールガイドの組み込み方法を参照し、コンパイラ本体を再インストールしてください。
3320	(F) COMMAND PARAMETER BUFFER OVERFLOW
	コマンドラインの指定が 65535 文字を超えています。
3321	(F) ILLEGAL ENVIRONMENT VARIABLE
	環境変数 CH38 の設定でファイル名の規約に反した指定をしているか、パス名の長さが 118 文字を超えています。
3322	(F) LACKING CPU SPECIFICATION
	CPU/動作モードの指定がされていません。cpu オプションまたは環境変数 H38CPU で CPU/動作モードを指定してください。
3323	(F) ILLEGAL ENVIRONMENT SPECIFIED
	コンパイラで使用する環境変数（CH38TMP、H38CPU）の指定に誤りがあります。
4000 ～ 4999	(ー)INTERNAL ERROR
	コンパイラの内部処理で何らかの障害が生じました。本コンパイラをお求めになった営業所あるいは代理店にエラーの発生状況をご連絡ください。

4.2 C ライブラリ関数のエラーメッセージ

ライブラリ関数の中には，ライブラリ関数を実行中にエラーが発生した場合，標準ライブラリのヘッダファイル <stddef.h> で定義しているマクロ errno にエラー番号を設定するものがあります。エラー番号には，対応するエラーメッセージが定義してあり，エラーメッセージを出力することができます。エラーメッセージを出力するプログラム例を以下に示します。

■例

```
#include          <stdio.h>
#include          <string.h>
#include          <stdlib.h>

main()
{
     FILE *fp;

     fp=fopen("file", "w");
     fp=NULL;

     fclose(fp);                        /* error occurred  */

     printf("%s＼n", strerror(errno));/* print error message */
}
```

■説明

(1) fclose 関数に値 NULL のファイルポインタを実引数として渡しているので，エラーとなります。このとき errno に対応するエラー番号が設定されます。

(2) strerror 関数は，エラー番号を実引数として渡すと，対応するエラーメッセージの文字列のポインタを返します。printf 関数の文字列出力指定によりエラーメッセージを出力します。

表 4.3 C **ライブラリ関数のエラーメッセージ一覧**

エラー番号	エラーメッセージ／説明	エラー番号を設定する関数
1100 (ERANGE)	DATA OUT OF RANGE **オーバフローが発生しました。**	frexp, ldexp, modf, ceil, floor, fmod, strtol, atoi, atol, perror, fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf
1102 (EDIV)	DIVISION BY ZERO **ゼロによる除算を行なっています。**	div, ldiv
1104 (ESTRN)	TOO LONG STRING **文字列の長さが** 512 **文字を超えています。**	strtol, strtod, atoi, atol, atof
1106 (PTRERR)	INVALID FILE POINTER **ファイルポインタの値に** NULL **ポインタ定数を指定しています。**	fclose, fflush, freopen, setbuf, setvbuf, fprintf, fscanf, printf**,** scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, fgetc, fgets, fputc, fputs, ungetc, fread, fwrite, fseek, ftell, rewind, perror
1200 (ECBASE)	INVALID RADIX **基数の指定が誤っています。**	strtol, atoi, atol
1202 (ETLN)	NUMBER TOO LONG **数値を表現する文字列の長さが** 17 **桁を超えています。**	strtod, fscanf, scanf, sscanf, atof
1204 (EEXP)	EXPONENT TOO LARGE **指数部の桁数が３桁を超えています。**	strtod, fscanf, scanf, sscanf, atof
1206 (EEXPN)	NORMALIZED EXPONENT TOO LARGE **文字列を一度** IEEE **規格の** 10 **進形式に正規化したとき指数部の桁数が３桁を超えています。**	strtod, fscanf, scanf, sscanf, atof
1208 (ENUM)	NOT A NUMBER **実引数に非数を指定しています。**	frexp, ldexp, modf, ceil, fabs, floor, fmod
1210 (EFLOATO)	OVERFLOW OUT OF FLOAT float **型の** 10 **進数値が**, float **型の範囲を超えています(オーバフロー)。**	strtod, fscanf, scanf, sscanf, atof
1220 (EFLOATU)	UNDERFLOW OUT OF FLOAT float **型の** 10 **進数値が**, float **型の範囲を超えています(アンダフロー)。**	strtod, fscanf, scanf, sscanf, atof

エラー番号	エラーメッセージ／説明	エラー番号を設定する関数
1230 (EOVER)	FLOATING POINT OVERFLOW **数値定数が，**double **型の範囲を超えています(オーバフロー)。**	strtod, fscanf, scanf, sscanf, atof
1240 (EUNDER)	FLOATING POINT UNDERFLOW **数値定数が，**double **型の範囲を超えています(アンダフロー)。**	strtod, fscanf, scanf, sscanf, atof
1300 (NOTOPN)	FILE NOT OPEN **ファイルがオープンされていません。**	fclose, fflush, setbuf, setvbuf, fprintf, fscanf, printf, scanf, vfprintf, vprintf, fgetc, fgets, fputc,fputs, gets, puts, ungetc, fread, fwrite, fseek, ftell, rewind, perror, freopen
1302 (EBADF)	BAD FILE NUMBER **入力専用ファイルに対して出力関数，あるいは出力専用ファイルに対して入力関数を発行しています。**	fprintf, fscanf, printf, scanf, sprintf, sscanf, vfprintf, vprintf, vsprintf, fgetc, fgets, fputc, fputs, gets, puts, ungetc, perror, fread, fwrite
1304 (ECSPEC)	ERROR IN FORMAT **書式付き入出力関数で指定している書式が誤っています。**	fprintf, fscanf, printf, scanf,sprintf, sscanf, vfprintf, vprintf, vsprintf, perror

5. モジュール間最適化

第5章　目次

5.1 概要 165
5.2 プログラムの開発手順 167
5.3 最適化ツールの起動方法 168
5.4 オプション／サブコマンド 171
5.4.1 最適化内容指定 173
5.4.2 最適化抑止指定 176
5.4.3 最適化情報 178
5.4.4 サブコマンドファイル 179

5.1 概要

H8S,H8/300 シリーズモジュール間最適化ツールは、H8S,H8/300 シリーズ用 C コンパイラおよび H8S,H8/300 シリーズ用アセンブラが出力した複数のオブジェクトプログラムを入力し、モジュールをまたがって最適化を実行した上で、結合および編集するソフトウェアシステムです。従来 C コンパイラだけでは最適化できなかった、メモリ配置や関数の呼び出し関係に依存した最適化を実現します。

本最適化ツールをご使用になる際には、以下のソフトウェアが必要です。

H8S,H8/300 シリーズ C コンパイラシステム（Ver2.0）

- H8S,H8/300 シリーズモジュール間最適化ツール Ver.1.0（本システム）
- H8S,H8/300 シリーズ C コンパイラ（Ver2.0 以降）
- CPU 情報ファイル作成プログラム(Ver1.0 以降）

H8S,H8/300 シリーズクロスアセンブラシステム（Ver2.0）*[1]

- H8S,H8/300 シリーズクロスアセンブラ（Ver2.0 以降）
- H シリーズリンケージエディタ（Ver5.3 以降）
- H シリーズオブジェクトコンバータ（V1.5 以降）

【注】 *1 H8S,H8/300 **アセンブラシステム（**Ver1.0 **台）をご使用の場合は、アセンブラ出力のオブジェクトプログラムは最適化の対象になりません。**

■注意

（1）本最適化ツールは、コンパイル時、アセンブル時に goptimize オプションを指定して付加情報ファイルを生成したオブジェクトプログラムを最適化の対象にします。
goptimize 指定なしで生成したオブジェクトプログラムや、H8S,H8/300 シリーズ C コンパイラ（Ver1.0 台）で生成したオブジェクトプログラムと、goptimize オプションを指定して生成したオブジェクトプログラムは、入力ファイルとして混在して指定できます。このとき最適化の対象となるのは、goptimize オプションを指定して生成したオブジェクトプログラムのみになります。

（2）C コンパイラ、アセンブラは、オブジェクトプログラム出力ディレクトリに「ch38iop」という名前のディレクトリを自動生成し、ディレクトリ「ch38iop」の下に付加情報ファイルを格納します。オブジェクトプログラムを移動する場合は、付加情報ファイルも一緒に移動してください。

（3）本 C コンパイラに添付されている SYSROF ファイルコンバータを用いて、旧形式のオブジェクトフォーマットに変換したオブジェクトプログラムは、本最適化ツールに入力することはできません。一旦全てのオブジェクトプログラム、ライブラリファイルを新形式に変換し、本最適化ツールでアブソリュートロードモジュールを生成して

ください。SYSROF ファイルコンバータの詳細については、製品添付の「H8S,H8/300 シリーズ SYSROF ファイルコンバータ 取扱説明書」を参照してください。

5.2 プログラムの開発手順

本最適化ツールを用いたプログラムの開発手順を**図** 5.1 に示します。

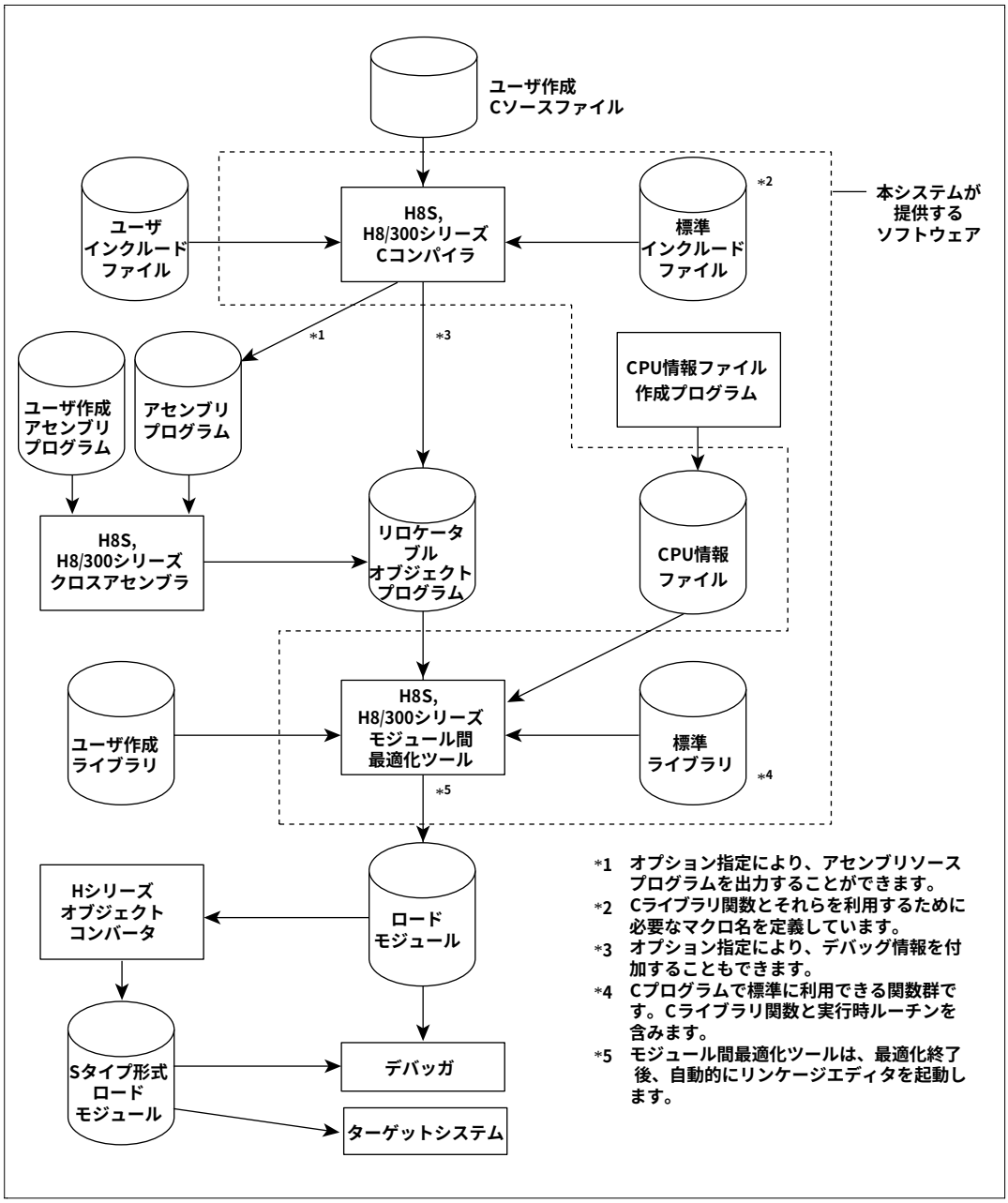


図 5.1 **プログラムの開発手順**

5.3 最適化ツールの起動方法

最適化ツールを起動するコマンドラインの形式は次のとおりです。

```
optlnk38[△<オプション>...]
```

最適化ツールを実行するためには、関連ソフトウエアを使用して、次のファイルを作成する必要があります。（括弧内は関連ソフトウエア名称）

オブジェクトプログラム（H8S,H8/300 シリーズ C コンパイラ、H8S,H8/300 シリーズ アセンブラ）
CPU 情報ファイル（CPU 情報ファイル作成プログラム）
リンケージエディタ用サブコマンドファイル

以下、最適化ツールの基本的な操作方法をサンプルプログラムを用いて説明します。
ここではサンプルプログラムとして、製品中の sample ディレクトリ内に含まれている次のファイルを使用します。

test1.c	C プログラム
test1.sub	リンケージエディタ用サブコマンドファイル
test2.sub	test1.sub に最適化ツール用オプションを追加
300ha.cpu	H8/300H アドバンストモード用 CPU 情報ファイル

（1）プログラムのコンパイル

test1.c をコンパイルします。このとき必ず goptimize オプションを指定します。また、ここで debug オプションを指定することにより、C ソースレベルデバッグを行うためのデバッグ情報を出力することができます。

```
ch38 -cpu=300ha -goptimize -debug test1 (RET)
```

（2）デフォルトライブラリの設定

リンク時に使用する標準ライブラリをデフォルトライブラリとして設定します。デフォルトライブラリについての詳細は「**H シリーズ リンケージエディタ、ライブラリアン、オブジェクトコンバータユーザーズマニュアル**」を参照してください。

PC 版： `set HLNK_LIBRARY1= <標準ライブラリパス>¥c38ha.lib (RET)`
UNIX 版： `setenv HLNK_LIBRARY1 <標準ライブラリディレクトリ>/c38ha.lib (RET)`

（3）最適化ツールの実行

test1.obj の最適化を実施後、ロードモジュールを作成します。

ここでは、リンケージエディタ用サブコマンドファイルの指定は省略できません。必ず指定してください。

■例 1：オブジェクトプログラムの最適化、結合

```
optlnk38 -optimize -subcommand=test1.sub (RET)
```

<test1.sub>

```
input   test1                ; 入力ファイル名を指定します
entry   _main                ; 実行開始の関数名を指定します
debug                        ; デバッグ情報出力を指定します
start   P,C(200),D,B(08000) ; 各セクションの開始アドレスを指定します
cpu     300ha.cpu            ; cpu 情報ファイル名を指定します
exit                         ; 処理を終了します
```

説明：

本最適化ツール起動時に指定するリンケージエディタ用サブコマンドファイルでは、cpu サブコマンドによる cpu 情報ファイルを必ず指定する必要があります。指定がない場合はエラーになります。

リンケージエディタ用サブコマンドファイルの作成方法は、「**H シリーズ リンケージエディタ、ライブラリアン、オブジェクトコンバータユーザーズマニュアル**」を参照してください。

■例 2：最適化オプションの指定

optimize オプションのサブオプションによって、最適化の内容を指定することができます。

```
optlnk38 -optimize=speed -subcommand=test1.sub (RET)
```

■例3：サブコマンドによる最適化オプションの指定

最適化ツールのオプションはサブコマンドとして、リンケージエディタ用サブコマンドファイル内で指定することもできます。

```
optlnk38 -subcommand=test2.sub (RET)
```

<test2.sub>

```
optimize  speed                ; 最適化ツール用オプションを指定します
input     test1                ; 入力ファイル名を指定します
entry     _main                ; 実行開始の関数名を指定します
debug                          ; デバッグ情報出力を指定します
start     P,C(200),D,B(08000) ; 各セクションの開始アドレスを指定します
cpu       300ha.cpu            ; cpu 情報ファイル名を指定します
exit                           ; 処理を終了します
```

（4）コマンド入力形式、オプションの表示

標準出力画面上にコマンドの入力形式、オプションの一覧を表示します。

```
optlnk38 (RET)
```

5.4 オプション／サブコマンド

オプション／サブコマンドの形式は次のとおりです。

オプション　　：　-<オプション>[=<パラメタ>[,<パラメタ>...]]
サブコマンド*　：　<サブコマンド>△[<パラメタ>[,<パラメタ>...]]

【注】 * サブコマンドファイルの記述規則および制限事項は、「Hシリーズ リンケージエディタ、ライブラリアン、オブジェクトコンバータユーザーズマニュアル」を参照してください。

オプション／サブコマンドと短縮形および省略時解釈の一覧を**表** 5.1 に示します。下線部（__）は短縮形指定時の文字を示します。

表 5.1 **オプション一覧**

No.	項目	オプション／サブコマンド名	パラメタ	省略時解釈	指定内容
1	**最適化内容**	optimize	string_unify | symbol_delete | variable_access | register | same_code | function_call | branch | speed | safe	optimize= string_unify, symbol_delete, variable_access, register, same_code, function_call, branch **（または** optimize**）**	**最適化内容の指定** **定数／文字列の統合** **未参照シンボルの削除** **短絶対アドレッシングモードの活用** **レジスタの再割付** **共通コードの統合** **間接アドレッシングモードの活用** **分岐命令の最適化** **スピード重視の最適化** (op=st,sy,v,r,b) **安全な最適化**(op=st,r,b)
		nooptimize	—	**なし**	**最適化の抑止指定**
		samesize	<size> size：16 **進数**	samesize=1E	**共通コード統合** (optimize=same_code)**の対象サイズ指定**

No.	項目	オプション／サブコマンド名	パラメタ	省略時解釈	指定内容
2	最適化抑止	symbol_forbid	<シンボル名> [,<シンボル名>...] シンボル名：<変数名>|<関数名>	なし	未参照シンボル削除(optimize=symbol_delete)の最適化を抑止する変数／関数名を指定
		samecode_forbid	<関数名> [,<関数名>...]	なし	共通コード統合(optimize=same_code)の最適化を抑止する関数名を指定
		variable_forbid	<変数名> [,<変数名>...]	なし	短絶対アドレッシングモード活用(optimize= variable_access)の最適化を抑止する変数名を指定
		function_forbid	<関数名> [,<関数名>...]	なし	間接アドレッシングモード活用(optimize= function_call)の最適化を抑止する関数名を指定
		absolute_forbid	<addr>[+<size>] [,<addr>[+<size>]...]	なし	アドレス割付の対象外となるアドレス領域を指定
3	最適化情報	information	—	なし	最適化された関数名の表示指定
4	サブコマンドファイル	subcommand	<ファイル名>	なし	リンケージエディタ用サブコマンドファイルの指定

5.4.1 最適化内容指定

（1）optimize オプション／サブコマンド

■形式

オプション： -optimize[=<パラメタ>[,<パラメタ>...]]

サブコマンド：optimize[△<パラメタ>[,<パラメタ>...]]

パラメタ： string_unify | symbol_delete | variable_access | register |same_code | function_call | branch | speed | safe

■説明

モジュール間最適化を実行します。また、パラメタを指定することにより、最適化内容を指定することができます。パラメタは複数指定することができます。

表 5.2 optimize **オプション／サブコマンドのパラメター覧**

パラメタ	説　明
パラメタなし	**全ての最適化を実行します。**optimize=string_unify, symbol_delete, variable_access, register, same_code, function_call, branch **を指定した時と同じです。**
string_unify	const **属性を持つ定数／文字列に対し、同一値定数および同一文字列の統合を、モジュール間に亘って実施します。この最適化は、**C **コンパイラ出力のオブジェクトプログラムに対してのみ実施します。**const **属性を持つ定数／文字列には、次のものが含まれます。** **・**C **プログラム中で** const **宣言した変数** **・文字列データの初期値**
symbol_delete	**一度も参照のない変数／関数を削除します。この最適化は、**C **コンパイラ出力のオブジェクトプログラムに対してのみ実施します。この最適化を指定する場合は、必ずリンケージエディタ用サブコマンドファイル内で** entry **サブコマンドを指定してください。**
variable_access	8 **ビットおよび** 16 **ビット絶対アドレッシングモードでアクセス可能な領域に空きがあれば、アクセス回数の多い変数を割り当てる最適化を行います。割り当てる対象となる変数は、**C **プログラム内で定義した変数のみですが、アクセスコードの最適化はアセンブリプログラム内での参照箇所も対象になります。**

パラメタ	説　明
register	関数の呼出関係を解析し、冗長なレジスタ退避・回復コードを削除します。また、呼出前後のレジスタ使用状況により、使用レジスタ番号を変更することもあります。この最適化は、C コンパイラ出力のオブジェクトプログラムに対してのみ実施します。
same_code	複数の同一命令列をサブルーチン化して、コードサイズを削減します。この最適化は、C コンパイラ出力のオブジェクトプログラムに対してのみ実施します。
function_call	0〜0xFF の範囲に空きがあれば、アクセス回数の多い関数のアドレスを割り当てる最適化を行います。この最適化は、C プログラム、アセンブリプログラムに関係なく対象になります。
branch	プログラムの配置情報に基づいて、分岐命令サイズを最適化します。この最適化は、C プログラム、アセンブリプログラムに関係なく対象になります。また、他の最適化項目をひとつでも実行すると、本最適化は指定の有無に関わらず、必ず実行します。
speed	最適化項目のうち、同一命令列のサブルーチン化のようなオブジェクトスピード低下を招く可能性のある最適化以外を実施します。optimize=speed は、optimize=string_unify, symbol_delete, variable_access, register,branch を指定したときと同じ効果になります。
safe	メモリ割り付け位置が固定でなければならない変数や、スピードを優先したい関数など、部分的に最適化を抑止したい場合があります。optimize=safe は、変数や関数の属性によって制限される可能性のある最適化以外を実施します。optimize=safe は、optimize=string_unify, register, branch を指定したときと同じ効果になります。

■注意

optimize オプション／サブコマンドのパラメタは、指定されたパラメタの論理和が有効となります。例えば、optimize=speed,same_code が指定された場合、

optimize= string_unify,symbol_delete,variable_access, register,branch,same_code が有効になります。すなわち、optimize=function_call 以外の全ての最適化を実施します。

（2）nooptimize オプション／サブコマンド

■形式

オプション　：　-nooptimize
サブコマンド　：　nooptimize
パラメタ　：　なし

■説明

モジュール間最適化を実行せずに、subcommand オプションで指定したリンケージエディタコマンドに従って、結合・編集作業のみ行い、リンケージエディタを直接起動した時と同じロードモジュールを生成します。

（3）samesize オプション／サブコマンド

■形式

オプション　：　-samesize=<パラメタ>
サブコマンド　：　samesize△<パラメタ>
パラメタ　：　<数値>

■説明

optimize=same_code で、最適化の対象となる共通コードのサイズを指定します。このオプションで指定するサイズは、オブジェクトプログラムの実際のバイト数を指します。optimize=same_code オプションが有効でない場合には、本オプションは無視されます。

■数値

16 進数で指定します。指定したサイズ以上の命令列について、共通コードのサブルーチン化を実施します。本オプション省略時は、samesize=1E を仮定します。指定できる範囲は 8≦ 数値 ≦ 7FFF です。

5.4.2 最適化抑止指定

（1）symbol_forbid オプション／サブコマンド

■形式

オプション　：　-symbol_forbid=<パラメタ>[,<パラメタ>...]
サブコマンド：　symbol_forbid△<パラメタ>[,<パラメタ>...]
パラメタ　　：　<シンボル名>

■説明

未参照シンボル削除（optimize=symbol_delete）の最適化を抑止する変数名／関数名を指定します。optimize=symbol_delete オプションが有効でない場合には、本オプション／サブコマンドは無視されます。

・シンボル名
　変数名、関数名はC プログラム中での定義名の先頭に_を付加します。
　最適化によって削除してはならない変数名、関数名を指定してください。

（2）samecode_forbid オプション／サブコマンド

■形式

オプション　：　-samecode_forbid=<パラメタ>[,<パラメタ>...]
サブコマンド：　samecode_forbid△<パラメタ>[,<パラメタ>...]
パラメタ　　：　<関数名>

■説明

共通コード統合（optimize=same_code）の最適化を抑止する関数名を指定します。optimize=same_code オプションが有効でない場合には、本オプション／サブコマンドは無視されます。

・関数名
　関数名はC プログラム中での定義名の先頭に_を付加します。
　共通コード統合の最適化は、コードサイズ削減には効果がありますが、実行速度が低下する可能性があります。スピードを重視する関数等、共通コード統合化を抑止したい関数名を指定してください。

（3）variable_forbid オプション／サブコマンド

■形式

オプション　：　-variable_forbid=<パラメタ>[,<パラメタ>...]
サブコマンド：　variable_forbid△<パラメタ>[,<パラメタ>...]
パラメタ　　：　<変数名>

■説明

短絶対アドレッシングモード活用（optimize=variable_access）の最適化を抑止する変数名を指定します。 optimize=variable_access オプションが有効でない場合には、本オプション／サブコマンドは無視されます。

・シンボル名

変数名はC プログラム中での定義名の先頭に_を付加します。
C プログラム内で定義し、アセンブリプログラム内で間接参照している等、アドレスを変更してはならない変数名を指定してください。

（4）function_forbid オプション／サブコマンド

■形式

オプション　：　-function_forbid=<パラメタ>[,<パラメタ>...]
サブコマンド：　function_forbid△<パラメタ>[,<パラメタ>...]
パラメタ　　：　<関数名>

■説明

間接アドレッシングモード活用（optimize=function_call）の最適化を抑止する関数名を指定します。optimize=function_call オプションが有効でない場合には、本オプション／サブコマンドは無視されます。

・関数名

関数名は C プログラム中での定義名の先頭に_を付加します。間接アドレッシングモードの活用は、コードサイズ削減には効果がありますが、実行速度が低下する可能性があります。スピードを重視する関数等、間接アドレッシングモードの使用を抑止したい関数名を指定してください。

（5）absolute_forbid オプション／サブコマンド

■形式

オプション　：　-absolute_forbid=<パラメタ>[,<パラメタ>...]
サブコマンド：　absolute_forbid△<パラメタ>[,<パラメタ>...]
パラメタ　　：　<アドレス>[+<size>]

■説明

最適化時に使用できないアドレス領域を指定します。

・アドレス[+サイズ]

アドレス、サイズとも 16 進数で指定します。また、アドレス値が A～Fの値で始まる場合は、先頭に 0 を付加してください。optimize=function_call オプションや optimize=variable_access オプションでは、オブジェクト性能を向上するために、積極的にメモリ割り付けの変更を行います。I/O 領域やエミュレータ予約領域など、最適化で使用できないアドレス領域を指定してください。

5.4.3 最適化情報

（1）information オプション／サブコマンド

■形式

オプション　：　-information
サブコマンド：　information
パラメタ　　：　なし

■説明

最適化された関数名の表示を指定します。

5.4.4 サブコマンドファイル

（1）subcommand オプション

■形式

オプション　：　-subcommand=<パラメタ>
パラメタ　　：　<ファイル名>

■説明

リンケージエディタ用サブコマンドファイルを指定します。

本最適化ツールは、最適化処理後、リンケージエディタを自動的に起動し、結合・編集処理を行います。そのため、本オプションを省略することはできません。

・ファイル名

リンケージエディタ用サブコマンドファイル名称を指定します。

サブコマンドファイル名には、-を含めることはできません。

■注意

リンケージエディタ用サブコマンドファイルおよび、サブコマンドに関する説明は、「H**シリーズリンケージエディタ、ライブラリアン、オブジェクトコンバータ ユーザーズマニュアル**」を参照してください。また、以下の注意事項に注意して、サブコマンドファイルを作成してください。

（a）cpu 情報ファイルは、最適化ツールがメモリ割り付け可能なアドレス領域を参照するために使用します。したがって、cpu サブコマンドによる cpu 情報ファイルの指定は省略することはできません。省略時にはエラーになります。cpu 情報ファイルの作成方法は、「**付録** G.　cpu **情報ファイルの作成**」を参照してください。

（b）subcommand オプションで指定するリンケージエディタ用サブコマンドファイルには、最適化ツール用サブコマンドを指定することができます。但し、最適化ツール用サブコマンドを指定したサブコマンドファイルを、Hシリーズリンケージエディタの subcommand オプションで指定した場合、エラーになります。

（c）print サブコマンドを指定した場合、マップリストには最適化ツール出力のテンポラリサブコマンドファイル名が出力されます。このテンポラリサブコマンドファイルは、最適化ツール終了後削除されます。

（d）form サブコマンドで指定できるパラメタは”a”のみです。

（e）リンケージエディタサブコマンドの delete,rename,exchange は使用できません。

6. モジュール間最適化のエラーメッセージ

第 6 章　目次

6.1　モジュール間最適化のエラーメッセージ 183

6.1 モジュール間最適化のエラーメッセージ

本章では、以下の形式でモジュール間最適化ツールの出力するエラーメッセージとエラー内容を説明します。

エラー番号	エラーメッセージ
	エラー内容

エラーレベルは、エラーの重要度に従い、４種に分類されます。

表 6.1 モジュール間最適化ツールのエラーレベル

0 ～ 999	インフォメーション	処理を継続し、ロードモジュールを出力します。
1000 ～ 1999	ウォーニング	処理を継続し、ロードモジュールを出力します。
2000 ～ 2999	エラー	処理を継続します。ロードモジュールは出力しません。
3000 ～ 3999	フェータル	処理を中断します。

表 6.2 モジュール間最適化ツールのエラーメッセージ

0010	<ユニット名 1> IS REPLACED WITH <ユニット名 2>(<ファイル名>)
	<ユニット名 1>を(ファイル名)中の<ユニット名 2>に置き換えました。
0020	<外部名 1> IS RENAMED TO <外部名 2>
	<外部名 1>を<外部名 2>に変更しました。
0030	<外部名> IS DELETED
	<外部名>を削除しました。
0040	DUPLICATE UNIT - (<ユニット名>) IN (<ファイル名>) IS DELETED
	<ユニット名>のユニットを複数見つけたため、<ファイル名>中のユニット名を削除しました。
0050	<外部参照シンボル名> CANNOT BE DEFINED
	<外部参照シンボル名>が見つからないため、強制定義できません。
0060	<外部名> CANNOT BE RENAMED
	<外部名>が見つからないため、変更できません。
0070	<外部名> CANNOT BE DELETED
	<外部名>が見つからないため、削除できません。
0080	<ユニット名> CANNOT BE REPLACED
	<ユニット名>が見つからないため、置き換えができません。
0200	<最適化種別> OPTIMIZE :<セクション名> SECTION IS CREATED
	<最適化種別>の最適化によって<セクション名>を作成しました。

0210	<最適化種別> OPTIMIZE:<ユニット名>.<シンボル名> MOVED <セクション名> SECTION
	<最適化種別>の最適化によって<ユニット名>.<シンボル名>を<セクション名>に移動しました。
0220	<最適化種別> OPTIMIZE:<ユニット名>.<シンボル名> IS CREATED
	<最適化種別>の最適化によって<ユニット名>.<シンボル名>を作成しました。
0230	<最適化種別> OPTIMIZE:<ユニット名>.<シンボル名> IS DELETED
	<最適化種別>の最適化によって<ユニット名>.<シンボル名>を削除しました。
0240	<ユニット名>.<シンボル名> IS OPTIMIZED
	<ユニット名>.<シンボル名>を最適化しました。
1010	DUPLICATE OPTION/SUBCOMMAND(<オプション／サブコマンド名>)
	同じオプションまたはサブコマンドを重複して指定しています。後に指定したオプションまたはサブコマンドが有効になります。
1020	IDENTIFIER CHARACTER EXCEEDS 251(<名前>)
	251 文字を超える名前（ユニット名、セクション名、シンボル名）を指定しています。251 文字までが有効になります。
1040	DUPLICATE SYMBOL(<シンボル名>)
	外部定義シンボルが重複しています。先に現われた外部定義シンボルが有効になります。
1050	UNDEFINED EXTERNAL SYMBOL(<ユニット名>.<シンボル名>)
	未定義の外部シンボルを参照しています。外部参照は無効になり、0 を仮定します。
1060	REDEFINED SYMBOL(<シンボル名>)
	定義済みのシンボルを DEFINE オプション／サブコマンドで定義しています。DEFINE オプション／サブコマンドの指定を無視します。
1070	SECTION ATTRIBUTE MISMATCH(<セクション名>)
	属性または境界調整数の異なる同名セクションを入力しました。別セクションとして扱います。
1080	RELOCATION SIZE OVERFLOW(<ユニット名>.<セクション名> - <オフセット値>)
	リロケーションの結果がリロケーションサイズを超えました。
1090	ENTRY POINT MULTIPLY DEFINED
	実行開始アドレスの指定があるオブジェクトモジュールを複数指定しています。先に現われた実効開始アドレスの指定が有効になります。

1110	DUPLICATE SECTION NAME(<セクション名>)
	オプション／サブコマンドで同一セクション名を指定しています。最初に指定したセクション名を有効にします。
1120	ILLEGAL CPU INFORMATION FILE FORMAT
	CPU 情報ファイルのファイル形式が正しくありません。
1130	CONFLICTING DEVICE TYPE
	入力オブジェクトモジュールの対象 CPU と異なる CPU 情報ファイルを指定しています。
1140	SECTION IS NOT IN SAME MEMORY AREA(<セクション名>:xxxx-yyyy)
	セクションが一つのメモリ領域に入りきらず、xxxx 番地から yyyy 番地が異なるメモリ領域に割り付けられています。
1150	INACCESSIBLE ADDRESS RANGE(<セクション名>)
	セクションが使用できない領域に割り付けられています。
1160	INVALID CPU OPTION/SUBCOMMAND
	ロードモジュールファイルをリロケータブル形式に指定して、CPU オプション／サブコマンドを指定しています。
1170	ADDRESS SPACE DUPLICATE
	セクションが重複しています。
1180	INVALID UDF OPTION/SUBCOMMAND
	出力ロードモジュール形式がアブソリュート指定に対し、NOUDF オプション／サブコマンドを指定しています。NOUDF オプション／サブコマンドを無視します。
1190	RELOCATION VALUE IS ODD(<ユニット名>.<セクション名>-<オフセット値>)
	ディスプレースメントに対するリロケーション結果が奇数になりました。最下位ビットを切り捨てます。
1200	START ADDRESS NOT SPECIFIED FOR SECTION(<セクション名>)
	START オプション／サブコマンドで指定していないセクションが存在します。
1210	CANNOT FIND SECTION(<セクション名>)
	指定したセクションが見つかりません。
1220	TOO LONG SUBCOMMAND LINE
	ディレクトリ名の置き換えで文字数が 511 文字を超えました。511 文字までを有効とします。

1230	TOO MANY DIRECTORY COMMANDS
	DIRECTORY サブコマンドで 16 個を超えたディレクトリを指定しています。16 個までを有効とします。
1240	NO DEBUG INFORMATION
	デバッグ情報の全くないファイルに対して DEBUG、SDEBUG オプション／サブコマンドを指定しています。コンパイル、アセンブル時にデバッグオプションを指定してください。
1600	INVALID SYMBOL_FORBID OPTION
	SYMBOL_FORBID の指定が無効です。
1610	INVALID SAMECODE_FORBID OPTION
	SAMECODE_FORBID の指定が無効です。
1620	INVALID VARIABLE_FORBID OPTION
	VARIABLE_FORBID の指定が無効です。
1630	INVALID FUNCTION_FORBID OPTION
	FUNCTION_FORBID の指定が無効です。
1640	INVALID ABSOLUTE_FORBID OPTION
	ABSOLUTE_FORBID の指定が無効です。
1700	CANNOT FIND SYMBOL SPECIFIED SYMBOL_FORBID(<シンボル名>)
	SYMBOL_FORBID で指定したシンボル名が見つかりません。
1710	CANNOT FIND SYMBOL SPECIFIED SAMECODE_FORBID(<シンボル名>)
	SAMECODE_FORBID で指定したシンボル名が見つかりません。
1720	CANNOT FIND SYMBOL SPECIFIED VARIABLE_FORBID(<シンボル名>)
	VARIABLE_FORBID で指定したシンボル名が見つかりません。
1730	CANNOT FIND SYMBOL SPECIFIED FUNCTION_FORBID(<シンボル名>)
	FUNCTION_FORBID で指定したシンボル名が見つかりません。
1800	<最適化種別> OPTIMIZE:SECTION OVERLAP
	<最適化種別>の最適化でサイズ増加により隣接するセクションと重複しました。<最適化種別>の最適化指定を無効にします。
1810	DIFFERENT SYMBOL ASSIGNED TO A GLOBAL REGISTER AMONG FILES (<シンボル名>:<レジスタ番号>)
	グローバルレジスタに割り付けるシンボル名、レジスタ番号がファイル間で異なります。

1820	STACK ACCESS SIZE OVERFLOW
	レジスタ最適化でスタックアクセスコードがコンパイラのスタック量制限値を超えました。レジスタ最適化指定を無視します。
1830	RELOCATION VALUE EXISTS IN BCC/BSR(<ユニット名>)
	<ユニット名>のアセンブリプログラムで BCC/BSR に未解決のシンボルがあります。<ユニット名>を最適化対象外にします。
2010	ILLEGAL SUBCOMMAND/OPTION
	不正なサブコマンド名（またはオプション名）を指定しています。
2020	SYNTAX ERROR
	指定されたサブコマンド（またはオプション）に構文上の不正があります。
2030	TOO LONG SUBCOMMAND LINE
	サブコマンドの長さが 511 文字を超えています。
2040	ILLEGAL SUBCOMMAND SEQUENCE
	サブコマンドの指定順序が不正です。
2070	ILLEGAL SECTION NAME(<セクション名>)
	不正なセクション名を指定しています。
2080	ILLEGAL SYMBOL NAME(<シンボル名>)
	不正なシンボル名を指定しています。
2100	TOO MANY INPUT FILES
	入力ファイル数が 256 個を超えています。
2110	CANNOT FIND FILE(<ファイル名>)
	指定したファイルが見つかりません。
2120	CANNOT FIND UNIT(<ユニット名>)
	指定したユニットが見つかりません。
2130	CANNOT FIND MODULE(<モジュール名>)
	指定したモジュールが見つかりません。
2140	DUPLICATE START ADDRESS SPECIFIED
	同じ先頭アドレスを重複して指定しています。
2170	SUBCOMMAND COMMAND IN SUBCOMMAND FILE
	サブコマンドファイル中に SUBCOMMAND サブコマンドを指定しています。
2190	INVALID ADDRESS(<アドレス>)
	指定したアドレスが CPU のアドレス範囲を超えています。
2200	TOO MANY ROM COMMANDS
	ROM サブコマンドで 64 組を超えたセクションを指定しています。

2210	CANNOT CREATE ABSOLUTE MODULE(<**モジュール名**>)
	未定義の外部参照シンボルが存在しています。
2220	DIVISION BY ZERO IN RELOCATION VALUE(<**ユニット名**>.<**セクション名**> . <**オフセット値**>)
	0 **除算を含むオブジェクトファイルを入力しました。**
2600	COMPILER SUPPLEMENTARY INFORMATION FILE MISMATCH(<**ファイル名**>)
	コンパイラ付加情報ファイルの作成日付がオブジェクトと一致しません。
2610	ILLEGAL DUPLICATE SYMBOL(<**シンボル名**>)
	外部定義シンボルが重複しています。
2700	ILLEGAL ADDRESS SPECIFIED ABSOLUTE_FORBID OPTION(<**アドレス**>+<**サイズ**>)
	絶対アドレスの最適化抑止指定範囲が CPU **のアドレス範囲を超えました。**
2730	ILLEGAL SAMESIZE SPECIFIED
	共通コードサイズ指定が正しくありません。
2740	CANNOT OPTIMIZE RELOCATABLE FILE
	出力ロードモジュールファイル形式にリロケータブルを指定しています。
2750	NOT SPECIFIED ENTRY SUBCOMMAND
	optimize=symbol_delete **を指定していますが、**entry **サブコマンド指定がありません。**
2760	<**サブコマンド名**> NOT SUPPORT
	該当サブコマンドはモジュール間最適化ツールではサポートしていません。一旦リンケージエディタでリロケータブルファイル出力後、再度 optlnk38 **を実行してください。**
3010	ILLEGAL COMMAND PARAMETER
	不正なコマンドパラメータを指定しています。
3020	CANNOT OPEN FILE(<**ファイル名**>)
	ファイルをオープンできません。
3030	CANNOT READ INPUT FILE(<**ファイル名**>)
	ファイルを読み込むことができません。
3040	CANNOT WRITE OUTPUT FILE(<**ファイル名**>)
	ファイルに書き込むことができません。
3050	CANNOT CLOSE FILE(<**ファイル名**>)
	ファイルをクローズできません。

3060	ILLEGAL FILE FORMAT(<ファイル名>)
	指定したファイルのフォーマットが不正です。または、RENAME サブコマンドで指定した外部シンボル名が既に存在します。
3070	ILLEGAL RECORD FORMAT(<ファイル名>)
	指定したファイル中に不正なレコードがあります。または、除数が 0 の除算があります。
3080	SECTION ADDRESS OVERFLOW(<セクション名>)
	セクションの割り付けアドレスが CPU で許されるアドレス範囲を超えています。
3090	ADDRESS OVERFLOW
	指定したアドレスが CPU で許されるアドレス範囲を超えています。
3100	MEMORY OVERFLOW
	最適化ツールが内部で使用するメモリ領域を割り当てることができません。
3110	PROGRAM ERROR(<nnn>)
	最適化ツールの内部処理で何らかの障害が発生しました。プログラムエラー番号(nnn)を確認の上、当社営業担当までご連絡ください。
3120	ILLEGAL START ADDRESS ALIGNMENT (<アドレス>)
	オブジェクトモジュールの境界調整数と矛盾するアドレスを指定しています。
3140	CANNOT FIND SECTION(<セクション名>)
	指定したセクションが見つかりません。
3190	AUTOPAGE SPECIFIED AT NON-PAGE TYPE
	非ページタイプの入力ファイルに対して AUTOPAGE オプション／サブコマンドを指定しています。
3220	PAGE ADDRESS SPECIFIED AT NON-PAGE TYPE
	非ページタイプの入力ファイルに対してページアドレスを指定しています。
3230	SECTION SPECIFIED AT ROM OPTION/SUBCOMMAND DOES NOT EXIST(<セクション名>)
	ROM コマンドで指定したセクションが存在しません。
3250	ILLEGAL START SECTION(<セクション名>)
	START コマンドで指定したセクションの属性が不正です。
3260	CANNOT READ
	指定したファイル（標準入力を含む）から入力できません。
3270	SYMBOL ADDRESS OVERFLOW(<シンボル名>)
	シンボルの割り付けアドレスが CPU のアドレス範囲を超えています。

3280	ILLEGAL ROM SECTION(<セクション名>)
	ROM オプション／サブコマンドの指定で、転送先セクションにサイズ 0 以外のセクションあるいは絶対番地セクションを指定しています。または、転送元と転送先のセクションの属性が異なっています。
3290	INVALID MEMORY MAP
	CPU 情報ファイルと矛盾したメモリに割り付けています。または、異なるメモリ種別にまたがって割り付けています。
3300	ILLEGAL FILE FORMAT (INPUT ABSOLUTE FILE)
	アブソリュートロードモジュールを入力ファイルに指定しています。
3310	ILLEGAL FILE FORMAT (MISMATCH OBJECT FORMAT VERSION)
	オブジェクト形式の異なるファイルを入力しました。
3320	ILLEGAL FILE FORMAT (INPUT MISMATCH CPU TYPE)
	H シリーズ、SH シリーズ以外のファイルを入力しました。
3700	CANNOT OPEN CPU INFORMATION FILE
	CPU 情報ファイルがオープンできません。
3710	CANNOT OPEN INTERNAL FILE
	中間ファイルがオープンできません。
3720	CANNOT WRITE INTERNAL FILE
	中間ファイルに書き込むことができません。
3730	CANNOT CLOSE INTERNAL FILE
	中間ファイルをクローズできません。
3740	CANNOT EXECUTE(<ロードモジュール名>)
	opt38 または lnk を起動できません。正しくインストールされているか確認してください。
3750	CANNOT CREATE INTERNAL FILE
	中間ファイルを作成することができません。
3760	INTERRUPT BY USER
	処理中に標準入力端末から「(CNTL) + C」コマンドによる割り込みを検出しました。
3770	CANNOT ANALYZE OBJECT(<ユニット名>)
	オブジェクトコードを解析することができません。プログラムセクション内の.DATA 制御命令を削除するか、または当該ユニットのアセンブル時の GOPTIMIZE オプション指定を外してください。

3800	TOO MANY EXTERNAL DEFINE SYMBOLS(<**ユニット名**>)
	ユニット内の定義シンボル数が 65535 **を超えました。ファイルを分割するか、または当該ユニットのコンパイル、アセンブル時の** GOPTIMIZE **オプション指定を外してください。**
3810	TOO MANY EXTERNAL REFERENCE SYMBOLS(<**ユニット名**>)
	ユニット内の参照シンボル数が 65535 **を超えました。ファイルを分割するか、または当該ユニットのコンパイル、アセンブル時の** GOPTIMIZE **オプション指定を外してください。**
3820	TOO MANY SECTIONS(<**ユニット名**>)
	ユニット内のセクション数が 65535 **を超えました。ファイルを分割するか、または当該ユニットのコンパイル、アセンブル時の** GOPTIMIZE **オプション指定を外してください。**
3830	<**最適化種別**> OPTIMIZE:SECTION OVERLAP
	<最適化種別>の最適化でサイズ増加により隣接するセクションと重複しました。

付　録

目　次

A.　C コンパイラが規定する言語仕様と　C ライブラリ関数仕様 195
A.1　言語仕様 195
A.2　C ライブラリ関数仕様 204
A.3　浮動小数点の仕様 209
B.　引数割り付けの具体例 218
B.1　H8S/2600 用、H8S/2000 用、H8/300H 用 218
B.2　H8/300 用 223
C.　レジスタとスタック領域の使用法 226
C.1　H8S/2600 用、H8S/2000 用、H8/300H 用アドバンストモード 226
C.2　H8S/2600 用、H8S/2000 用、H8/300H 用ノーマルモード 228
C.3　H8/300 用 230
D.　終了処理関数の作成例 231
D.1　終了処理の登録と実行（onexit）ルーチンの作成例 231
D.2　プログラムの終了（exit）ルーチンの作成例 231
D.3　異常終了（abort）ルーチンの作成例 234
E.　低水準インタフェースルーチンの作成例 235
E.1　低水準インタフェースルーチン作成例 235
F.　短絶対アドレスのアクセス範囲 242
F.1　短絶対アドレスのアクセス範囲 242
G.　CPU 情報ファイルの作成 243
G.1　CIA の機能 243
G.2　CIA の起動方法 243
G.3　CIA の使用手順と選択メニュー 244
G.4　CIA の使用例 246
G.5　制限事項一覧 247
G.6　CIA のエラーメッセージ 248
H.　旧バージョンとの相違点 249

H.1　機能追加・改善項目 249
H.2　言語仕様の拡張 251
I.　ASCII コード一覧表 254
I.1　ASCII コード一覧表 254

A.　C コンパイラが規定する言語仕様と C ライブラリ関数仕様

A.1　言語仕様

C 言語仕様（H シリーズ C 言語マニュアル）で規定していない処理系定義項目について、C コンパイラの仕様を示します。

(1) 翻　訳

表 A.1　翻訳の仕様

No.	項　　目	C コンパイラの仕様
1	エラー検出時のエラー情報	「4.　C コンパイラのエラーメッセージ」を参照。

(2) 環　境

表 A.2　環境の仕様

No.	項　　目	C コンパイラの仕様
1	main 関数への実引数の意味	規定しません。
2	対話的入出力装置の構成	規定しません。

(3) 識別子

表 A.3　識別子の仕様

No.	項　　目	C コンパイラの仕様
1	外部結合とならない識別子（内部名）の有効文字数	先頭から 31 文字まで有効です。
2	外部結合となる識別子（外部名）の有効文字数	先頭から 31 文字まで有効です。
3	外部結合となる識別子（外部名）の大文字と小文字の区別	大文字と小文字を区別します。

■注意

31 文字目までが同じで、32 文字目以降が異なる二つの識別子は、同じ識別子とみなします。

■例

（a）　longnameabcdefghijklmnopqrstuvwx;

（b）　longnameabcdefghijklmnopqrstuvwy;

（a）と（b）の二つの識別子は、31 文字目までが一致しているので、同じ識別子とみなされます。

(4) 文　字

表 A.4　**文字の仕様**

No.	項　　目	C コンパイラの仕様
1	ソース文字集合および実行環境文字集合の要素	どちらも ASCII 文字集合です。ただし、文字列、文字定数にはシフト JIS または EUC 漢字コードを記述できます。
2	多バイト文字のコード化で使用されるシフト状態	シフト状態はサポートしていません。
3	プログラム実行時の文字集合の文字のビット数	ビット数は 8 ビットです。
4	文字定数内､文字列内のソース文字集合の文字と実行環境文字集合の文字との対応付け	同じ ASCII 文字に対応します。
5	言語で規定していない文字や拡張表記を含む整数文字定数の値	言語で規定する以外の文字、拡張表記はサポートしていません。
6	2 文字以上の文字を含む文字定数または 2 文字以上の多バイト文字を含む広角文字定数の値	文字定数は上位 2 文字を有効とします。広角文字定数は上位 1 文字を有効とします。また、1 文字より多く指定した場合はウォーニングエラーを出力します。
7	多バイト文字を広角文字に変換するために使用される locale の仕様	locale はサポートしていません。
8	単なる char 型が signed char 型と同じ値の範囲を持つか、unsigned char 型と同じ値の範囲を持つか	signed char 型と同じ値の範囲を持ちます。

(5) 整数

表 A.5　整数の仕様

No.	項　　目	C コンパイラの仕様
1	整数型の表現方法とその値	表 A.6 に示します。
2	整数の値がより短いサイズの符号付き整数型、または符号なし整数型を同一のサイズの符号付き整数型に変換したときの値（結果の値が変換先の型で表現できない場合）	整数の値の下位 2 バイトあるいは下位 1 バイトが変換後の値になります。
3	符号付き整数に対するビットごとの演算の結果	符号付きの値になります。
4	整数除算における剰余の符号	被除数の符号と同符号になります。
5	負の値を持つ符号付き汎整数型の右シフトの結果	符号ビットを保持します。

表 A.6　整数型とその値の範囲

No.	型	値　の　範　囲	データサイズ
1	char	−128〜127	1 バイト
2	signed char	−128〜127	1 バイト
3	unsigned char	0〜255	1 バイト
4	short	−32768〜32767	2 バイト
5	unsigned short	0〜65535	2 バイト
6	int	−32768〜32767	2 バイト
7	unsigned int	0〜65535	2 バイト
8	long	−2147483648〜2147483647	4 バイト
9	unsigned long	0〜4294967295	4 バイト

（6）浮動小数点

表 A.7　浮動小数点の仕様

No.	項　　目	C コンパイラの仕様
1	浮動小数点型の表現方法とその値	浮動小数点型には、float 型、double 型と long double 型があります。浮動小数点型の内部表現や変換仕様、演算仕様等の性質は「A.3　浮動小数点数の仕様」で説明します。表 A.8 に、浮動小数点型の表現可能な値の限界値を示します。
2	整数を本来の値に正確に表現することができない浮動小数点数に変換したときの切り捨て方向	
3	浮動小数点数をより狭い浮動小数点数に変換したときの切り捨てまたは丸め方法	

表 A.8　浮動小数点数の限界値

No.	項　　目	限　界　値	
		10 進数表現　*1	16 進数表現
1	float 型の最大値	3.4028235677973364e＋38f (3.4028234663852886e＋38f)	7f7fffff
2	float 型の正の最小値	7.0064923216240862e－46f (1.4012984643248171e－45f)	00000001
3	double, long double 型の最大値	1.7976931348623158e＋308 (1.7976931348623157e＋308)	7fefffffffffffff
4	double, long double 型の正の最小値	4.9406564584124655e－324 (4.9406564584124654e－324)	0000000000000001

【注】　*1 10 進数表現の限界値は 0 または無限大にならない限界値です。また、（）内は理論値を示します。

（7）配列とポインタ

表 A.9　配列とポインタの仕様

No.	項　　目	C コンパイラの仕様
1	配列の大きさの最大値を保持するために必要な整数の型（size_t）	unsigned int 型　H8S/2600 用ノーマルモード、H8S/2000 用ノーマルモード、H8/300H 用ノーマルモード、H8/300 用
		unsigned long 型　H8S/2600 用アドバンストモード、H8S/2000 用アドバンストモード、H8/300H 用アドバンストモード
2	ポインタ型から整数型への変換（ポイント型のサイズ≧整数型のサイズ）	ポインタ型の下位バイトの値になります。
3	ポインタ型から整数型への変換（ポイント型のサイズ＜整数型のサイズ）	0 拡張します。
4	整数型からポインタ型への変換（整数型のサイズ≧ポイント型のサイズ）	整数型の下位バイトの値となります。
5	整数型からポインタ型への変換（整数型のサイズ＜ポイント型のサイズ）	0 拡張します。
6	同じ配列内のメンバのポインタ間の差を保持するために必要な整数の型（ptrdiff_t）	int 型　H8S/2600 用ノーマルモード、H8S/2000 用ノーマルモード、H8/300H 用ノーマルモード、H8/300 用
		long 型　H8S/2600 用アドバンストモード、H8S/2000 用アドバンストモード、H8/300H 用アドバンストモード

(8) レジスタ

表 A.10　レジスタの仕様

No.	項　　目	C コンパイラの仕様
1	レジスタ変数を割り付けることができるレジスタ	H8S/2600、H8S/2000、H8/300H 最適化あり（ER3）、ER4、ER5、ER6 最適化なし（ER3）、ER4、ER5 *1
		H8/300　最適化あり（R3）、R4、R5、R6 最適化なし（R3）、R4、R5 *1
2	レジスタに割り付けることができるレジスタ変数の型（H8S/2600、H8S/2000、H8/300H）	char、unsigned char、short、 unsigned short、int、 unsigned int、long、 unsigned long、float、 ポインタ
3	レジスタに割り付けることができるレジスタ変数の型（H8/300）	char、unsigned char、short、 unsigned short、int、 unsigned int、ポインタ

【注】　*1 （　）内のレジスタは noregexpansion オプションを指定した時は、レジスタ変数を割り付けません。

（9）構造体、共用体、列挙型、ビットフィールド

表 A.11　**構造体、共用体、列挙型、ビットフィールドの仕様**

No.	項　　目	C コンパイラの仕様
1	異なる型のメンバでアクセスされる共用体型のメンバ参照	参照はできますが、値は保証しません。
2	構造体メンバの境界整合	char 型のメンバだけからなる構造体は、1 バイト整合となります。それ以外は、2 バイト整合となります。割り付け方の詳細な仕様は「2.2.2（2）複合型」を参照してください。
3	単なる int 型のビットフィールドの符号	signed int 型とします。
4	int 型のサイズ内のビットフィールドの割り付け順序	上位ビットから割り付けます。
5	int 型のサイズ内にビットフィールドが割り付けられているとき、次に割り付けるビットフィールドのサイズが int 型内の残っているサイズをこえたときの割り付け方	次の int 型の領域に割り付けます。
6	ビットフィールドで許される型指定子	char、unsigned char、 short、unsigned short、 int、unsigned int
7	列挙型の値を表現する整数型	int 型

ビットフィールドの割り付け方の詳細については、**「2.2.2（3）ビットフィールド」**を参照してください。

（10）修飾子

表 A.12　**修飾子の仕様**

No.	項　　目	C コンパイラの仕様
1	volatile 型データへのアクセスの種類	規定しません。

（11）宣言

表 A.13　**宣言の仕様**

No.	項　　目	C コンパイラの仕様
1	**基本型（算術型、構造体型、共用体型）を修飾する宣言子の数**	16 **個まで指定できます。**

（a）基本型を修飾する型の数の数え方を、以下に例を用いて示します。

■例

（i）int a；a は int 型（基本型）であり、基本型を修飾する型の数は 0 個です。

（ii）char *f ( )；f は char 型（基本型）へのポインタ型を返す関数型であり、基本型を修飾する型の数は 2 個です。

（12）文

表 A.14　**文の仕様**

No.	項　　目	C コンパイラの仕様
1	**一つの** switch **文中で指定できる** case **ラベルの数**	511 **個まで指定できます。**

(13) プリプロセッサ

表 A.15　プリプロセッサの仕様

No.	項　　目	C コンパイラの仕様
1	条件コンパイルの定数式内の単一文字の文字定数と実行環境文字集合の対応	プリプロセッサ文の文字定数と実行環境文字集合は一致します。
2	インクルードファイルの読み込み方法	「〈」、「〉」で囲まれたファイルは include オプションで指定されたパスから読み込みます（省略時は環境変数 CH38 で設定されたパス）。
3	二重引用符で囲まれたインクルードファイルのサポートの有無	サポートします。インクルードファイルを現在のディレクトリから読み込みます。現在のディレクトリにない場合は No.2 の読み込み方法に従って読み込みます。
4	ソースファイルの文字の並びの対応（マクロ展開後の文字列の空白文字）	空白文字列は、空白文字 1 文字として展開します。
5	#pragma 文の動作	#pragma abs8、#pragma abs16、 #pragma asm、#pragma endasm、 #pragma indirect、 #pragma inline、 #pragma inline_asm、 #pragma section、 #pragma abs8 section、 #pragma abs16 section、 #pragma indirect section、 #pragma global_register、 #pragma regsave、 #pragma noregsave をサポートしています。
6	_ _DATE_ _、_ _TIME_ _の値	コンパイル開始時のホストマシンのタイマに基づく値が設定されます。

A.2　C ライブラリ関数仕様

C 言語仕様で規定されていない処理系定義の C ライブラリ関数仕様を以下に示します。

（1）stddef.h

表 A.16　stddef.h **の仕様**

No.	項　　目	C コンパイラの仕様
1	**マクロ** NULL **の値**	void **型へのポインタ型の値** 0 **です。**
2	**マクロ** ptrdiff_t **の内容**	int **型** H8S/2600 **用ノーマルモード 、** H8S/2000 **用ノーマルモード、** H8/300H **用ノーマルモード、** H8/300 **用**
		long **型** H8S/2600 **用アドバンストモード 、** H8S/2000 **用アドバンストモード、** H8/300H **用アドバンストモード**

（2）assert.h

表 A.17　assert.h **の仕様**

No.	項　　目	C コンパイラの仕様
1	assert **関数が出力する情報と終了動作**	**出力情報の形式を（**a**）に示します。情報を出力したあと** abort **関数を呼び出して終了します。**

（a）assert（式）において、式の値が 0 のとき以下のメッセージを出力します。

ASSERTION FAILED：Δ〈式〉ΔFILEΔ〈ファイル名〉, LINEΔ〈行番号〉

（3）ctype.h

表 A.18　ctype.h **の仕様**

No.	項　　目	C コンパイラの仕様
1	isalnum **関数、**isalpha **関数、**iscntrl **関数、**islower **関数** isprint **関数、**isupper **関数で検査される文字集合**	unsigned char **型で表現できる文字集合です。検査の結果真になる文字を表** A.19 **に示します。**

表 A.19　**真となる文字の集合**

No.	関　数　名	真　と　な　る　文　字
1	isalnum	'0' ～ '9'、'A' ～ 'Z'、'a' ～ 'z'
2	isalpha	'A' ～ 'Z'、'a' ～ 'z'
3	iscntrl	'＼X00' ～ '＼X1f'、'＼X7f'
4	islower	'a' ～ 'z'
5	isprint	'＼X20' ～ '＼X7E'
6	isupper	'A' ～ 'Z'

（4）math.h

表 A.20　math.h **の仕様**

No.	項　　目	C コンパイラの仕様
1	**数学関数の入力引数値が範囲をこえたときの数学関数が返す値**	**入力引数値が範囲をこえる数学関数はサポートしていません。**
2	**数学関数でアンダフローエラーが発生したときマクロ「**ERANGE**」の値が「**errno**」に設定されるかどうか**	**アンダフローエラーが発生する数学関数はサポートしていません。**
3	fmod **関数で第** 2 **実引数の値が** 0 **の場合、範囲エラーとなるかどうか**	**範囲エラーとなります。**

■注意

math.h には、C ライブラリー関数のエラー番号の値を示すマクロ EDOM、ERANGE が定義されています。

（5）stdio.h

表 A.21　stdio.h の仕様

No.	項　　目	C コンパイラの仕様
1	入力テキストの最終の行が終了を示す改行文字を必要とするかどうか	規定しません。低水準インタフェースルーチンの仕様によります。
2	改行文字の直前に書き出された空白文字は、読み込み時に読み込まれるかどうか	
3	バイナリファイルに書かれたデータに付加されるヌル文字の数	
4	追加モード時のファイル位置指定子の初期値	
5	テキストファイルへの出力によってそれ以降のファイルのデータが失われるかどうか	
6	ファイルバッファリングの仕様	
7	長さ 0 のファイルが存在するかどうか	
8	正当なファイル名の構成規則	
9	同時に同じファイルをオープンできるかどうか	
10	fprintf 関数における%p 書式変換の出力形式	16 進数出力となります。
11	fscanf 関数における%p 書式変換の入力形式 fscanf 関数での変換文字「－」の意味	16 進数入力となります。先頭、最後あるいは「＾」の直後でない場合、直前の文字と直後の範囲を示します。
12	fgetpos、ftell 関数で設定される errno の値	fgetpos 関数はサポートしていません。 ftell 関数については規定しません。低水準インタフェースルーチンの仕様によります。
13	perror 関数が生成するメッセージの出力形式	メッセージの出力形式を（a）に示します。
14	calloc、malloc、realloc 関数でサイズが 0 の時の動作	0 バイトの領域を割り付けます。

（a）perror 関数の出力形式は、
〈文字列〉：〈error に設定したエラー番号に対応するエラーメッセージ〉
となります。

（b）printf 関数、fprintf 関数等で、浮動小数点数の無限大および非数を表示するときの形式を**表** A.22 に示します。

表 A.22　**無限大および非数の表示形式**

No.	**値**	**表　示　形　式**
1	**正の無限大**	＋＋＋＋＋＋
2	**負の無限大**	－－－－－－
3	**非　　数**	＊＊＊＊＊＊

（6）string.h

表 A.23　string.h **の仕様**

No.	**項　　目**	**C コンパイラの仕様**
1	strerror **関数が返すエラーメッセージの内容**	**「**4.2　C **ライブラリ関数のエラーメッセージ」を参照してください。**

（7）サポートしていないライブラリ

H シリーズ C 言語マニュアルで定義している C 言語仕様のうち、本 C コンパイラでサポートしていないライブラリを**表** A.24 に示します。

表 A.24　**サポートしていないライブラリ**

No.	**ヘッダファイル**	**ラ　イ　ブ　ラ　リ　名**
1	math.h	acos、asin、atan、atan2、cos、sin、tan、cosh、sinh、tanh、exp、log、log10、pow、sqrt
2	signal.h	signal.h、signal、raise
3	stdio.h	remove、rename、tmpfile、tmpnam
4	stdlib.h	abort、exit、getenv、onexit、system
5	time.h	time.h、clock、difftime、time、asctime、ctime、gmtime、localtime

(8) 追加ライブラリ

H シリーズ C 言語マニュアルで定義しているライブラリに、以下のライブラリを追加しました。

<table>
<tr><td colspan="5">memmove</td></tr>
<tr><td>機　　能</td><td colspan="4">複写元の記憶域の内容を、指定した大きさ分、複写先の記憶域に複写します。また、複写元と複写先の記憶域が、重なっている部分があっても、複写元の重なっている部分を上書きする前に複写するので正しく複写されます。</td></tr>
<tr><td>呼び出し手　順</td><td colspan="4">#include <string.h>
void *ret, *s1;
const void *s2;
size_t　　n;
　　ret=memmove(s1,s2,n) ;</td></tr>
<tr><td rowspan="4">パラメタ</td><td>No.</td><td>名前</td><td>型</td><td>意　　味</td></tr>
<tr><td>1</td><td>s1</td><td>void 型を指すポインタ</td><td>複写先の記憶域へのポインタ</td></tr>
<tr><td>2</td><td>s2</td><td>const void 型を指すポインタ</td><td>複写元の記憶域へのポインタ</td></tr>
<tr><td>3</td><td>n</td><td>size_t</td><td>複写する文字数</td></tr>
<tr><td rowspan="3">リターン値</td><td>型</td><td colspan="3">void 型へのポインタ</td></tr>
<tr><td>正常</td><td colspan="3">s1 の値</td></tr>
<tr><td>異常</td><td colspan="3"></td></tr>
</table>

A.3　浮動小数点の仕様

(1) 浮動小数点数の内部表現

C コンパイラで扱う浮動小数点数の内部表現は、IEEE の標準形式に従っています。ここでは、IEEE 形式の浮動小数点数の内部表現の概要について述べます。

(a) 内部表現の形式

float型はIEEEの単精度形式（32 ビット）、double型と long double型はIEEEの倍精度形式（64 ビット）で表現します。

(b) 内部表現の構成

float型および double型と long double型の内部表現の構成を**図** A.1 に示します。

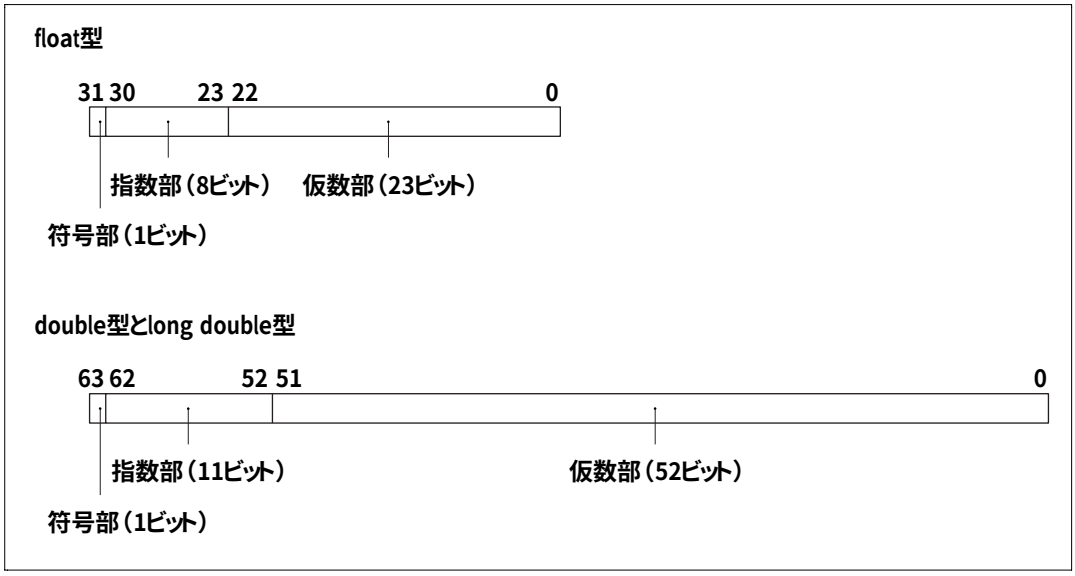


図 A.1　**浮動小数点数の内部表現の構成**

内部表現の各構成要素の意味を以下に示します。

（i）符号部

浮動小数点数の符号を示します。0 のとき正、1 のとき負を示します。

（ii）指数部

浮動小数点数の指数を 2 のべき乗で示します。

（iii）仮数部

浮動小数点数の有効数字に対応するデータです。

(c) 表現する値の種類

浮動小数点数は、通常の実数値のほかに、無限大等の値も表現することができます。浮動小数点数が表現する値の種類を以下に示します。

(i) 正規化数

指数部が 0 または全ビット 1 ではない場合です。通常の実数値を表現します。

(ii) 非正規化数

指数部が 0 で、仮数部が 0 でない場合です。絶対値の小さな実数値を表現します。

(iii) ゼロ

指数部および仮数部が 0 の場合です。値 0.0 を表現します。

(iv) 無限大

指数部が全ビット 1 で仮数部が 0 の場合です。無限大を表現します。

(v) 非数

指数部が全ビット 1 で仮数部が 0 でない場合です。「0.0/0.0」、「∞／∞」、「∞－∞」等、結果が数値に対応しない演算の結果として得られます。

浮動小数点数の表現する値を決定する条件を**表** A.25 に示します。

■注意

非正規化数は、正規化数で表現できない範囲の絶対値の小さな浮動小数点数を表現しますが、正規化数に比較して有効桁数が少なくなっています。したがって、演算の結果あるいは途中結果が非正規化数となる場合、結果の有効桁数は保証しませんので注意してください。

表 A.25　**浮動小数点数の表現する値の種類**

仮数部	指数部		
	0	0 でも全ビット 1 でもない	全ビット 1
0	0	正規化数	無限大
0 以外	非正規化数		非　数

（2）float 型

float型の内部表現は、1 ビットの符号部、8 ビットの指数部、23 ビットの仮数部からなります。

（i）正規化数

符号部は、0（正）または 1（負）で、値の符号を示します。

指数部は、1～254（2^8-2）の値をとります。実際の指数は、この値から 127 を引いた値で、その範囲は－126～127 です。

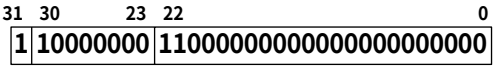
仮数部は、0～$2^{23}-1$ の値をとります。実際の仮数は、2^{23} のビットを 1 と仮定し、その直後に小数点があるものとして解釈します。

正規化数の表現する値は、

$$(-1)^{<符号部>} \times 2^{<指数部>-127} \times (1+<仮数部> \times 2^{-23})$$

となります。

■例



符号：　－

指数：　10000000(2)－127＝1

仮数：　1.11(2)＝1.75

値　：　$-1.75\times2^1=-3.5$

(2)は2進数を意味します。

（ii）非正規化数

符号部は 0（正）または 1（負）で、値の符号を示します。

指数部は 0 で、実際の指数は－126 になります。

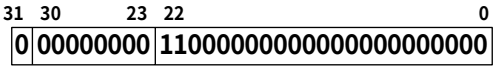
仮数部は、1～$2^{23}-1$ で、実際の仮数は、2^{23} のビットを 0 と仮定し、その直後に小数点があるものとして解釈します。

非正規化数の表現する値は、

$$(-1)^{<符号部>} \times 2^{-126} \times (<仮数部> \times 2^{-23})$$

となります。

■例



符号：　＋

指数：　－126

仮数：　$0.11_{(2)}=0.75$

値　：　0.75×2^{-126}

(2)は2進数を意味します。

（iii）ゼロ

符号部は 0（正）または 1（負）で、それぞれ＋0.0、－0.0 を示します。

指数部、仮数部はともに 0 です。

＋0.0、－0.0 は、ともに値としては 0.0 を示します。ゼロの符号による、各演算での機能の違いについては**「A.3（4）浮動小数点演算の仕様」**を参照してください。

（iv）無限大

符号部は 0（正）または 1（負）で、それぞれ＋∞、－∞を示します。

指数部は 255（2^8-1）です。

仮数部は 0 です。

（v）非数

指数部は 255（2^8-1）です。

仮数部は 0 以外の値です。

■注意

非数の符号、および仮数部の（0 以外の）値については規定していません。

（3）double 型と long double 型

double 型と long double 型の内部表現は、1 ビットの符号部、11 ビットの指数部、52 ビットの仮数部からなります。

（i）正規化数

符号部は 0（正）または 1（負）で、値の符号を示します。

指数部は、1～2046（$2^{11}-2$）の値をとります。実際の指数は、この値から 1023 を引いた値で、その範囲は－1022～1023 です。

仮数部は、0～$2^{52}-1$ の値となります。実際の仮数は、2^{52} のビットを 1 と仮定し、その直後に小数点があるものとして解釈します。

正規化数の表現する値は、

$$(-1)^{<符号部>} \times 2^{<指数部>-1023} \times (1 + <仮数部> \times 2^{-52})$$

となります。

■例

63	62 ～ 52	51 ～ 0
0	01111111111	11100000000000000000000000000000000000000000000000000

符号：　＋

指数：　$1111111111_{(2)}-1023=0$

仮数：　$1.111_{(2)}=1.875$

値　：　$1.875\times 2^0=1.875$

(2)は2進数を意味します。

（ii）非正規化数

符号部は 0（正）または 1（負）で、値の符号を示します。

指数部は 0 で、実際の指数は－1022 になります。

仮数部は、1～$2^{52}-1$ で、実際の仮数は、2^{52} のビットを 0 と仮定し、その直後に小数点があるものとして解釈します。

非正規化数が表現する値は、

$$(-1)^{<符号部>} \times 2^{-1022} \times (<仮数部> \times 2^{-52})$$

となります。

■例

63	62 … 52	51 … 0
1	00000000000	11100000000000000000000000000000000000000000000000000

符号：　－

指数：　－1022

仮数：　0.111(2)＝0.875

値　：　0.875×2^{-1022}

(2)は2進数を意味します。

（iii）ゼロ

符号部は 0（正）または 1（負）で、それぞれ＋0.0、－0.0 を示します。

指数部、仮数部はともに 0 です。

＋0.0、－0.0 は、ともに値としては 0.0 を示します。ゼロの符号による、各演算での機能の違いについては**「A.3（4）浮動小数点演算の仕様」**を参照してください。

（iv）無限大

符号部は 0（正）または 1（負）で、それぞれ＋∞、－∞を示します。

指数部は 2047（$2^{11}-1$）です。

仮数部は 0 です。

（v）非数

指数部は 2047（$2^{11}-1$）です。

仮数部は 0 以外の値です。

■注意

非数の符号、および仮数部の（0 以外の）値については規定していません。

（4）浮動小数点演算の仕様

本項では、C 言語の機能として表現されている浮動小数点の四則演算、およびコンパイル時や C ライブラリ関数の処理で生じる浮動小数点の 10 進表現と内部表現の間の変換の仕様について解説します。

（a）四則演算の仕様

（i）結果の値の丸め方

浮動小数点の四則演算の結果の正確な値が、内部表現の仮数の有効数字をこえた場合は、以下の規則に従って丸めを行ないます。

［1］ 結果の値は、その値を近似する二つの浮動小数点数の内部表現のうち、近い方に向かって丸めます。

［2］ 結果の値が、その値を近似する二つの浮動小数点数のちょうど中央になる場合は、仮数の最後の桁が 0 となる方向に丸めます。

（ii）オーバフロー、アンダフロー、無効演算時の処理

実行時のオーバフロー、アンダフロー、無効演算に対しては、以下の処理を行ないます。

［1］オーバフローの場合は、結果の符号に従って正または負の無限大になります。

［2］アンダフローの場合は、結果の符号に従って正または負のゼロになります。

［3］無効演算は、符号が逆の無限大を加算した場合、符号が同じ無限大を減算した場合、ゼロと無限大を乗算した場合、ゼロをゼロで、あるいは無限大を無限大で除算した場合に生じます。
これらの場合、結果は非数になります。

［4］上記のいずれの場合も、エラーの発生を示す変数 errno に対応するエラーの番号を設定します。この番号については、**「4.2　C ライブラリ関数のエラーメッセージ」**を参照してください。
エラーチェックが必要な場合は、この errno の値によってエラーの発生を判定してください。

■注意

定数式に関しては、コンパイル時に演算を行ないます。この時にオーバフロー、アンダフロー、無効演算を検出した場合は、ウォーニングレベルのエラーになります。

（iii）特殊値の演算に関する注意事項

以下、特殊な値（ゼロ、無限大、非数）の演算に関する注意事項を述べます。

［1］正のゼロと負のゼロの和は正のゼロとなります。

［2］同符号のゼロの差は正のゼロになります。

［3］被演算子の一方あるいは両方に非数を含む演算の結果は、常に非数になります。

［4］比較演算においては、正のゼロと負のゼロは等しいものとして扱います。

［5］被演算子の一方あるいは両方が非数であるような比較演算、等値演算の結果は、「！＝」については常に真、その他は常に偽となります。

（b）10 進表現と内部表現の間の変換

本項ではソースプログラム上の浮動小数点定数と内部表現の間の変換、あるいは C ライブラリ関数による ASCII 文字列による浮動小数点数の 10 進表現と、内部表現の間の変換の仕様について解説します。

（i）10 進表現から内部表現に変換する場合、まず 10 進表現を 10 進表現の正規形に変換します。

10 進表現の正規形は、「$\pm M \times 10^{\pm N}$」の形式で、M、N の範囲は以下のとおりです。

［1］ float 型の正規形

$0 \leqq M \leqq 10^{9}-1$

$0 \leqq N \leqq 99$

［2］ double 型と long double 型の正規形

$0 \leqq M \leqq 10^{17}-1$

$0 \leqq N \leqq 999$

正規形に変換できない 10 進表現については、オーバフロー、またはアンダフローになります。また、10 進表現が正規形よりも多くの有効数字を含んでいる場合は、下位の桁は切り捨てます。これらの場合、コンパイル時にはウォーニングレベルのエラーになり、実行時には対応するエラーの番号を変数 errno に設定します。

また、正規形に変換するためには、もとの 10 進表現の ASCII 文字列としての長さが 511 文字以下でなければなりません。そうでない場合、コンパイル時にはエラーになり、実行時には対応するエラーの番号を変数 errno に設定します。

内部表現から 10 進表現に変換する場合には、一度 10 進表現の正規形に変換してから、指定した書式に従って ASCII 文字列に変換します。

（ii）10 進表現の正規形と内部表現の間の変換

10 進表現の正規形と内部表現の間の変換は、指数が大きいときや小さいときには、正確な変換ができません。以下に、正確な変換ができる範囲と、その範囲外の場合の誤差の限界値について解説します。

（イ）正確な変換ができる範囲

以下に示す指数の範囲の浮動小数点数については、**「（a）（i）結果の値の丸め方」**に示す丸めが正確に行なわれます。この範囲ではオーバフロー、アンダフローは生じません。

float 型の場合： $0 \leqq M \leqq 10^{9}-1$、$0 \leqq N \leqq 13$

double 型と long double 型の場合： $0 \leqq M \leqq 10^{17}-1$、$0 \leqq N \leqq 27$

（ロ）誤差の限界値

（イ）で示す範囲に入っていない値を変換する場合の誤差と、正確な丸めを行なったときの誤差の差は、有効数字の最小位桁の 0.47 倍をこえません。

また、（イ）で示した範囲をこえている場合、変換の際にオーバフローやアンダフローが生じる場合があります。この場合、コンパイル時にはウォーニングレベルのエラーになり、実行時には対応するエラーの番号を変数 errno に設定します。

B. 引数割り付けの具体例

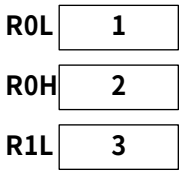
B.1 H8S/2600 用、H8S/2000 用、H8/300H 用

（cpu＝2600a、cpu＝2600n、cpu＝2000a、cpu＝2000n、cpu＝300ha、cpu＝300hn）

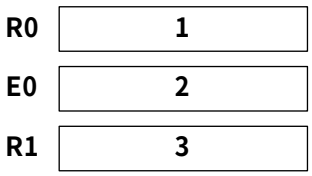
■例 1

レジスタ渡しの対象の型である引数は、宣言順にレジスタ ER0、ER1 に割り付けます。

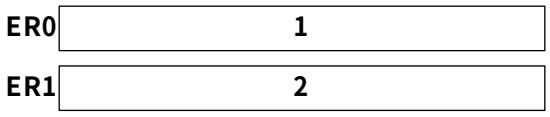
[1]
```
int f(char,char,char);
        :
    f(1,2,3);
        :
```



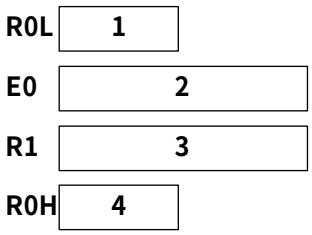
[2]
```
int f(int,int,int);
        :
    f(1,2,3);
        :
```



[3]
```
int f(long,long);
        :
    f(1,2);
        :
```



[4]
```
int f(char,int,int,char);
        :
    f(1,2,3,4);
        :
```

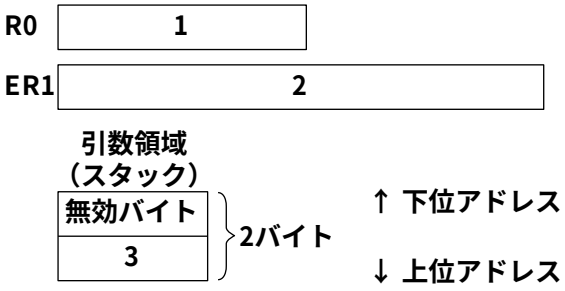


■例2

レジスタに割り付けることができなかった引数は、スタックに割り付けます。

また、引数の型がchar型で、スタック上の引数領域に割り付けた場合、下位アドレスに無効バイトができます。

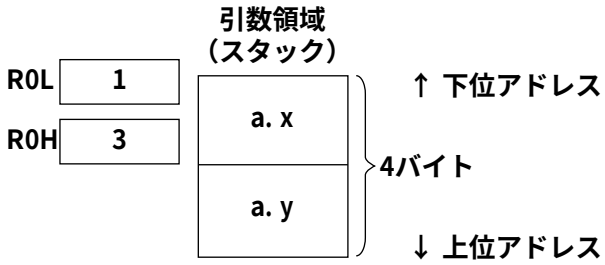
```
int f(int,long,char);
        :
      f(1,2,3);
        :
```



■例3

レジスタに割り付けられない型の引数は、スタックに割り付けます。

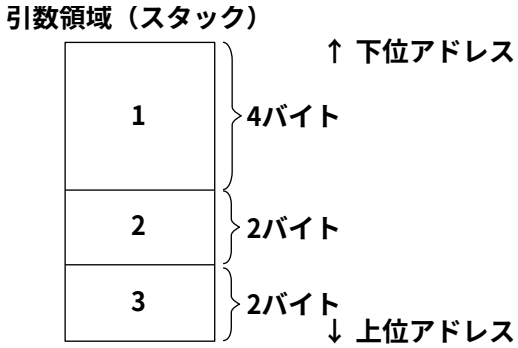
```
struct s{int x,y;)a;
int f(char,struct s,char);
     :
   f(1,a,3);
     :
```



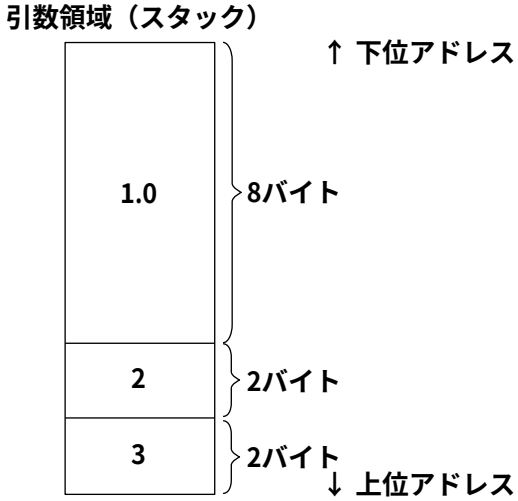
■例4

原型宣言により可変個の引数を持つ関数として宣言している場合、対応する型のない引数およびその直前の引数は、宣言順にスタックに割り付けます。

[1]
```
int f(long,...);
      :
    f(1,2,3);
      :
```



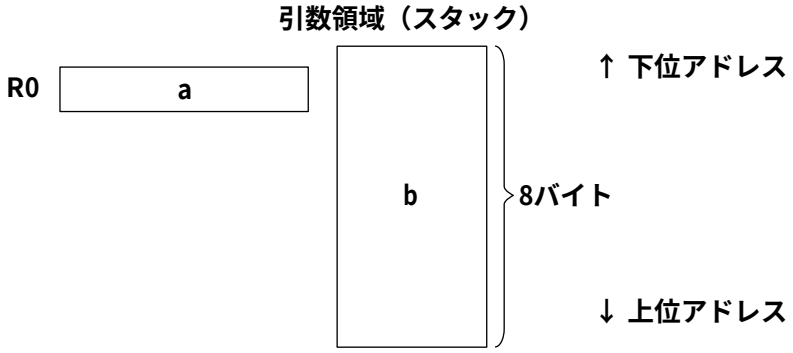
[2]
```
int f(double,int,...);
      :
    f(1.0,2,3);
      :
```



■例5

原型宣言がない場合、char 型は int 型に、float 型は double 型に拡張して渡します。

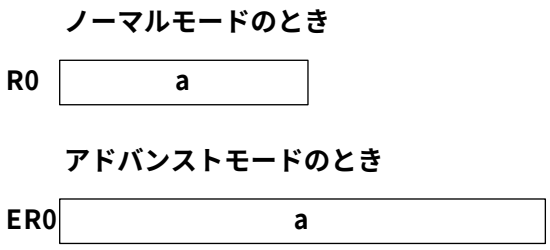
```
int f( );
char a;
float b;
          :
        f(a,b);
          :
```



■例6

ポインタ型は、ノーマルモードでは2バイト、アドバンストモードでは4バイトの領域に割り付けられます。

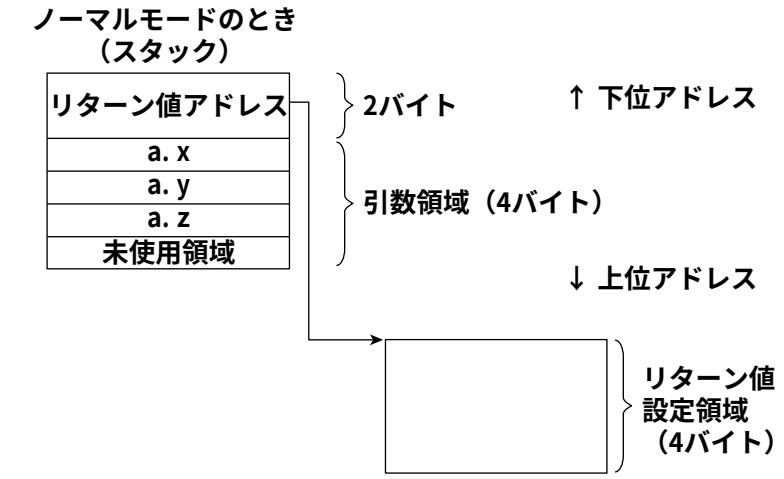
```
int f(int *);
          :
        f(a);
          :
          :
```

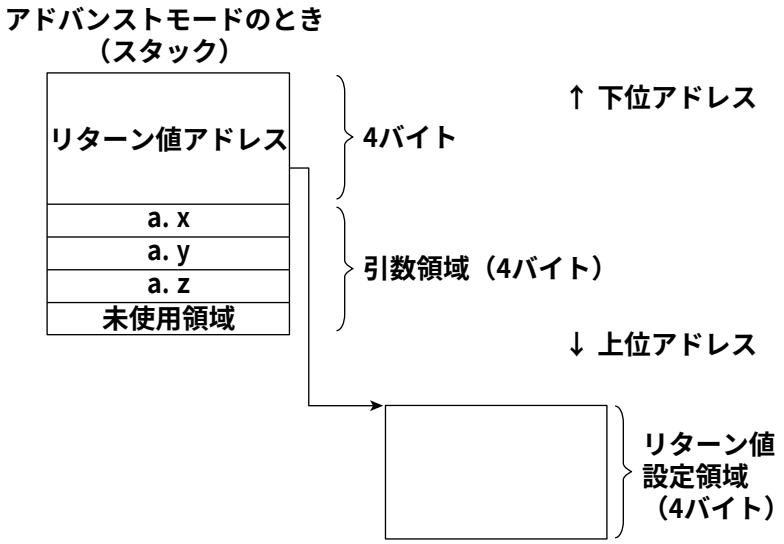


■例7

関数の返す型が4バイトをこえる場合または構造体の場合、引数領域の直前にリターン値アドレスを設定します。また、構造体のサイズが奇数バイトのとき、1バイトの未使用領域が生じます。

```
struct s{char x,y,z;}a,b;
float f(struct s);
          :
         f(a);
          :
          :
```



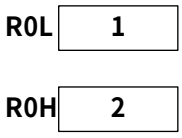


B.2　H8/300 用（cpu＝300）

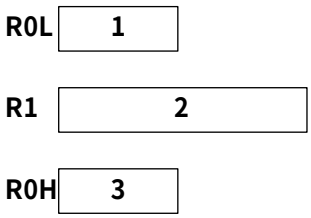
■例 1

レジスタ渡しの対象の型である引数は、宣言順にレジスタ R0、R1 に割り付けます。

[1]
```
int f(char,char);
        :
      f(1,2);
        :
```



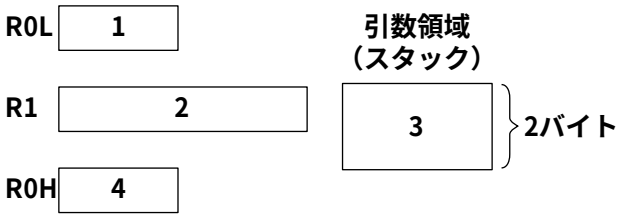
[2]
```
int f(char,int,char);
        :
      f(1,2,3);
        :
```



■例 2

レジスタに割り付けることができなかった引数は、スタックに割り付けます。

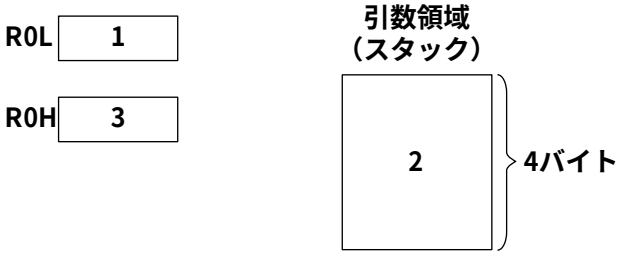
```
int f(char,int,int,char);
     :
   f(1,2,3,4);
     :
```



■例 3

レジスタに割り付けられない型の引数は、スタックに割り付けます。

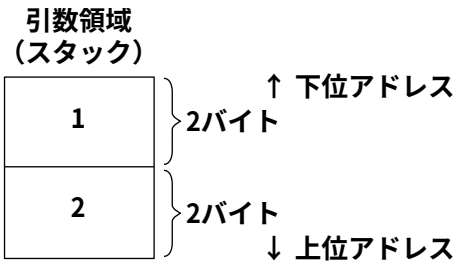
```
int f(char,long,char);
        :
      f(1,2,3);
        :
```



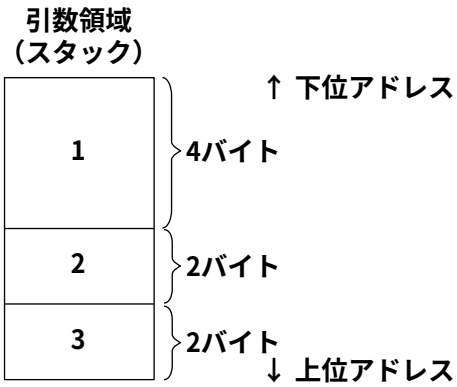
■例4

原型宣言により可変個の引数を持つ関数として宣言している場合、対応する型のない引数およびその直前の引数は、宣言順にスタックに割り付けます。

[1]
```
int f(int,...);
      :
    f(1,2);
      :
```



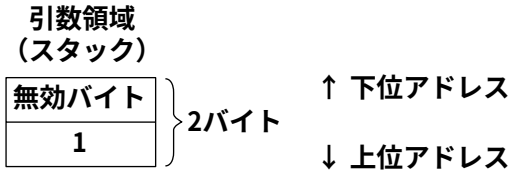
[2]
```
int f(long,int,...);
      :
    f(1,2,3);
      :
```



■例5

引数の型がchar型で、スタック上の引数領域に割り付けた場合、下位アドレスに無効バイトができます。

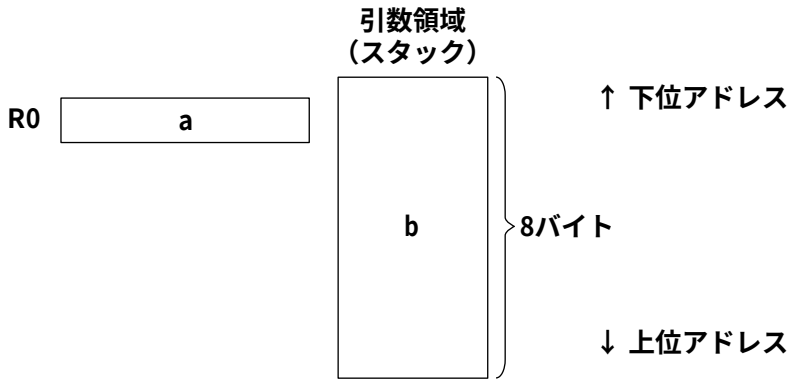
```
int f(char,...);
      :
    f(1);
      :
```



■例6

原型宣言がない場合、char型はint型に、float型はdouble型に拡張して渡します。

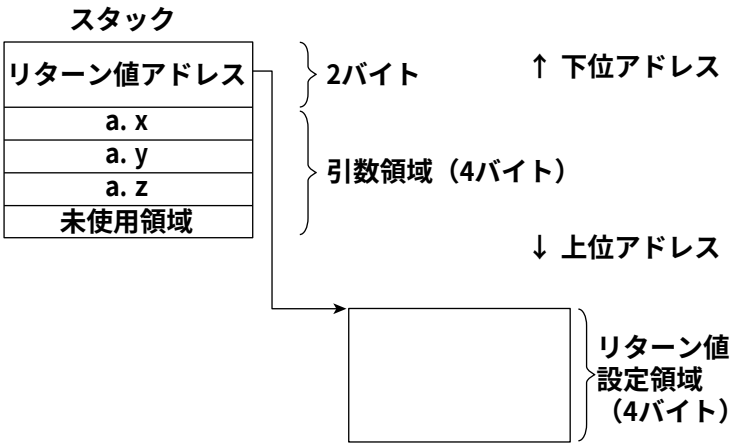
```
int f( );
char a;
float b;
          :
        f(a,b);
          :
```



■例7

関数の返す型が2バイトをこえる場合、引数領域の直前にリターン値アドレスを設定します。また、構造体のサイズが奇数バイトのとき、1バイトの未使用領域が生じます。

```
struct s{char x,y,z;}a,b;
float f(struct s);
           :
         f(a);
           :
           :
```



C. レジスタとスタック領域の使用法

C.1 H8S/2600用、H8S/2000用、H8/300H用アドバンストモード（cpu＝2600a、cpu＝2000a、cpu＝300ha）

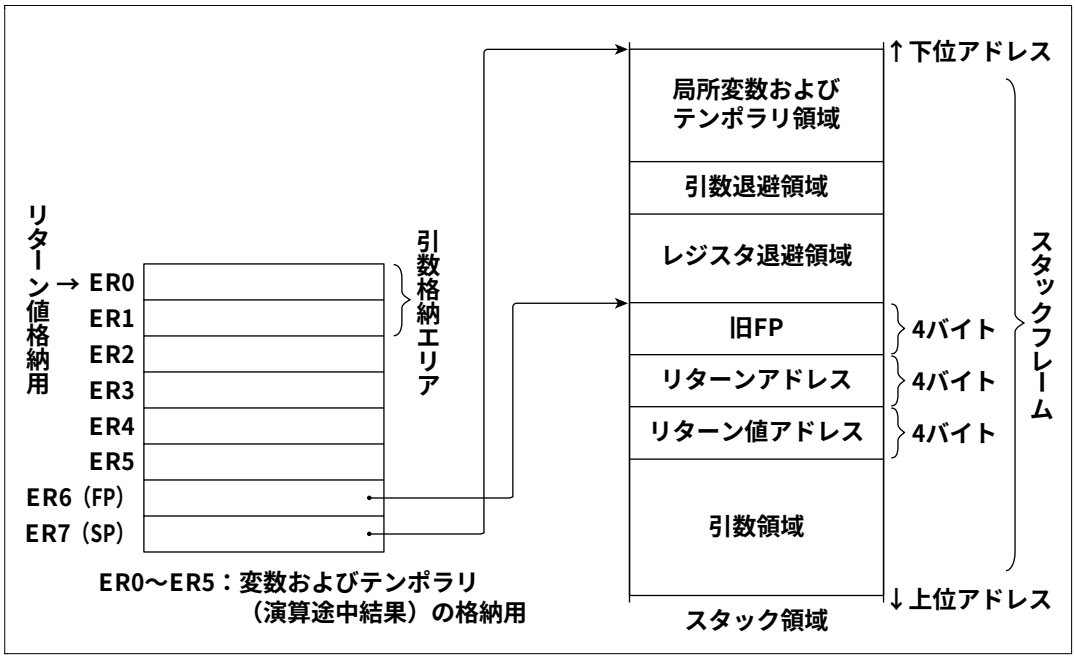


図 C.1 **非最適化時のレジスタとスタック領域の使用法**
（H8S/2600**用、**H8S/2000**用、**H8/300H**用アドバンストモード）**

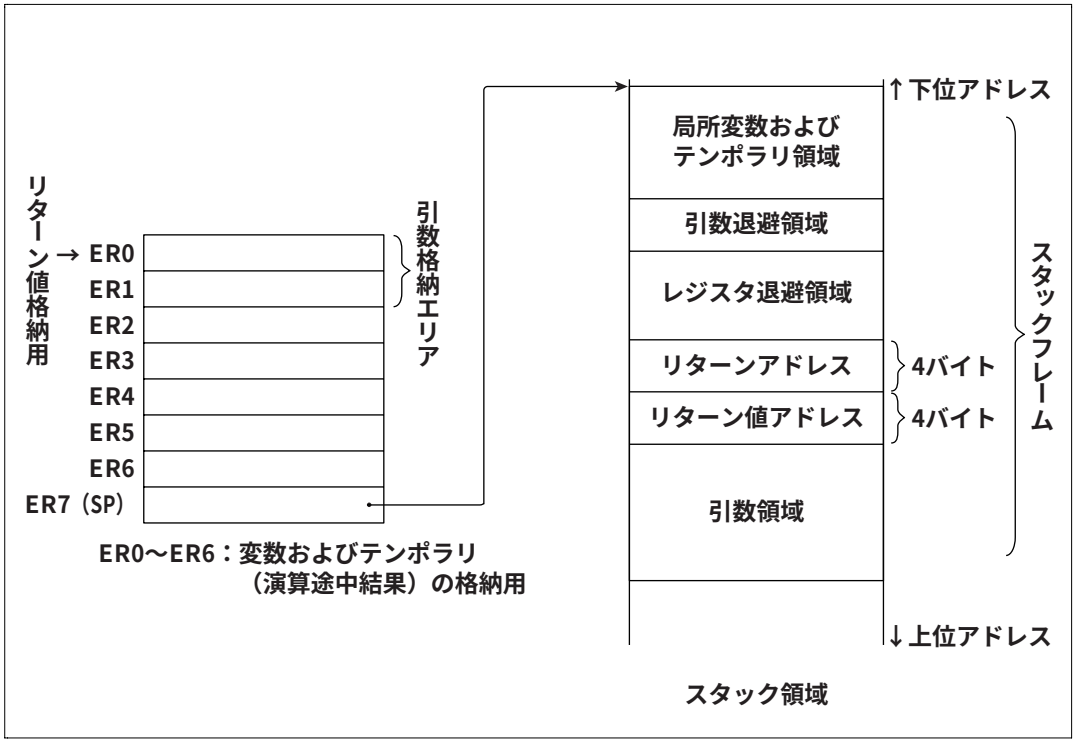


図 C.2　**最適化時のレジスタとスタック領域の使用法**

（H8S/2600 **用、**H8S/2000 **用、**H8/300H **用アドバンストモード）**

C.2 H8S/2600 用、H8S/2000 用、H8/300H 用ノーマルモード（cpu＝2600n、cpu＝2000n、cpu＝300hn）

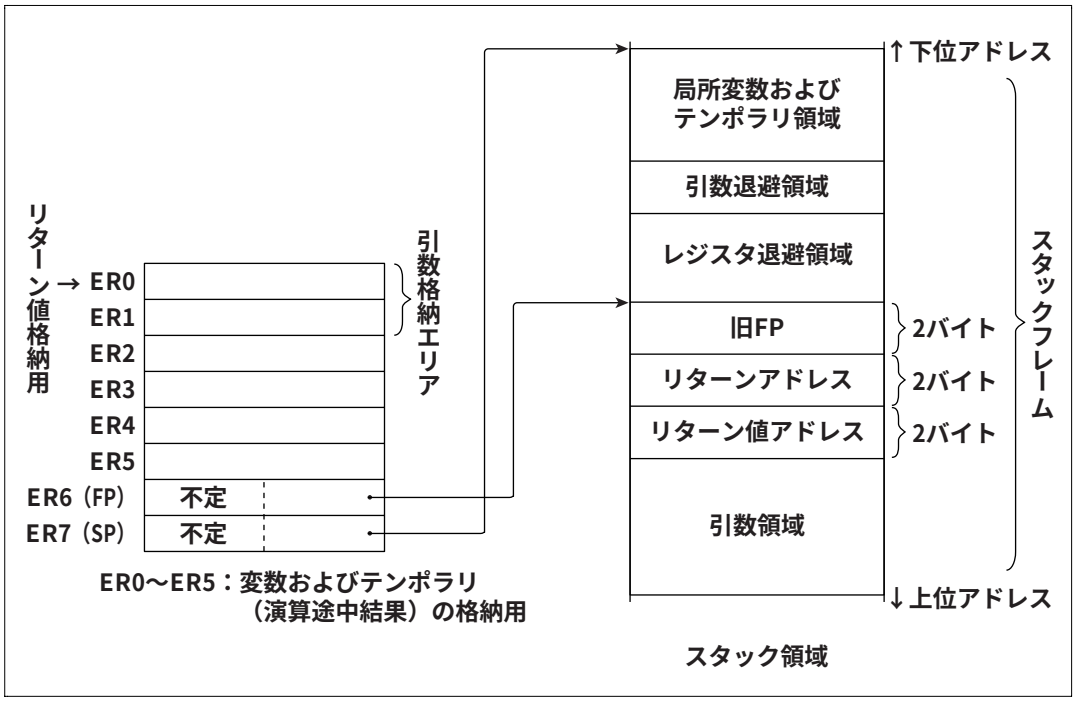


図 C.3 **非最適化時のレジスタとスタック領域の使用法**
（H8S/2600 **用、**H8S/2000 **用、**H8/300H **用ノーマルモード）**

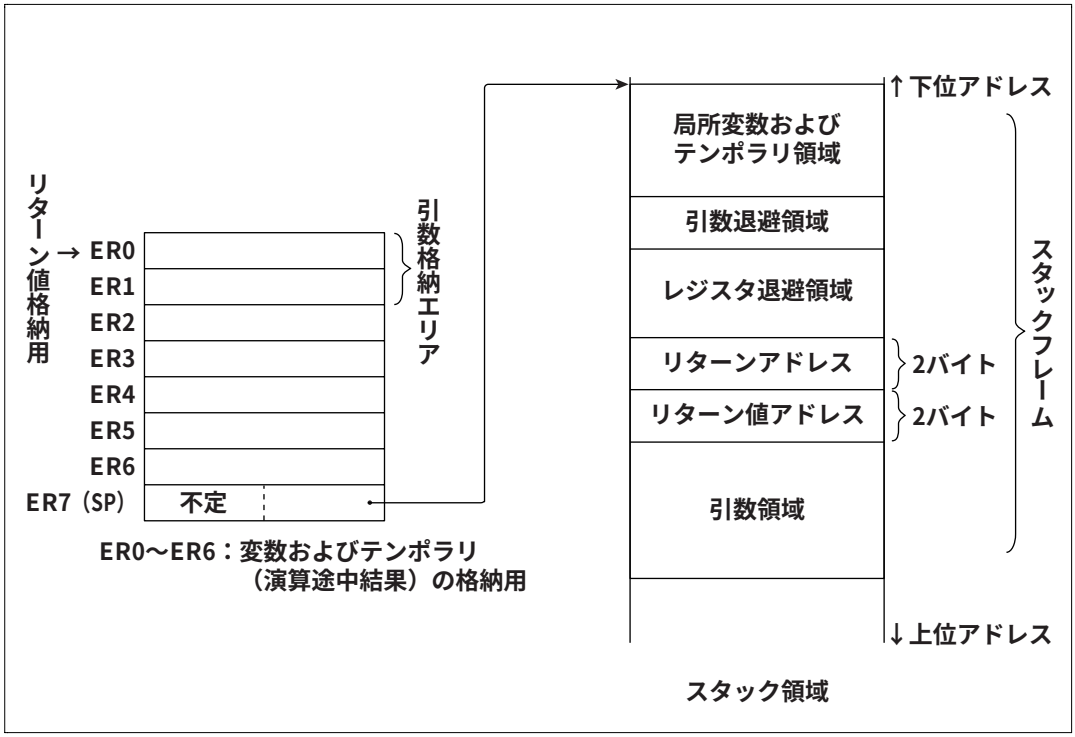


図 C.4　**最適化時のレジスタとスタック領域の使用法**

（H8S/2600 **用、**H8S/2000 **用、**H8/300H **用ノーマルモード）**

C.3　H8/300 用（cpu＝300）

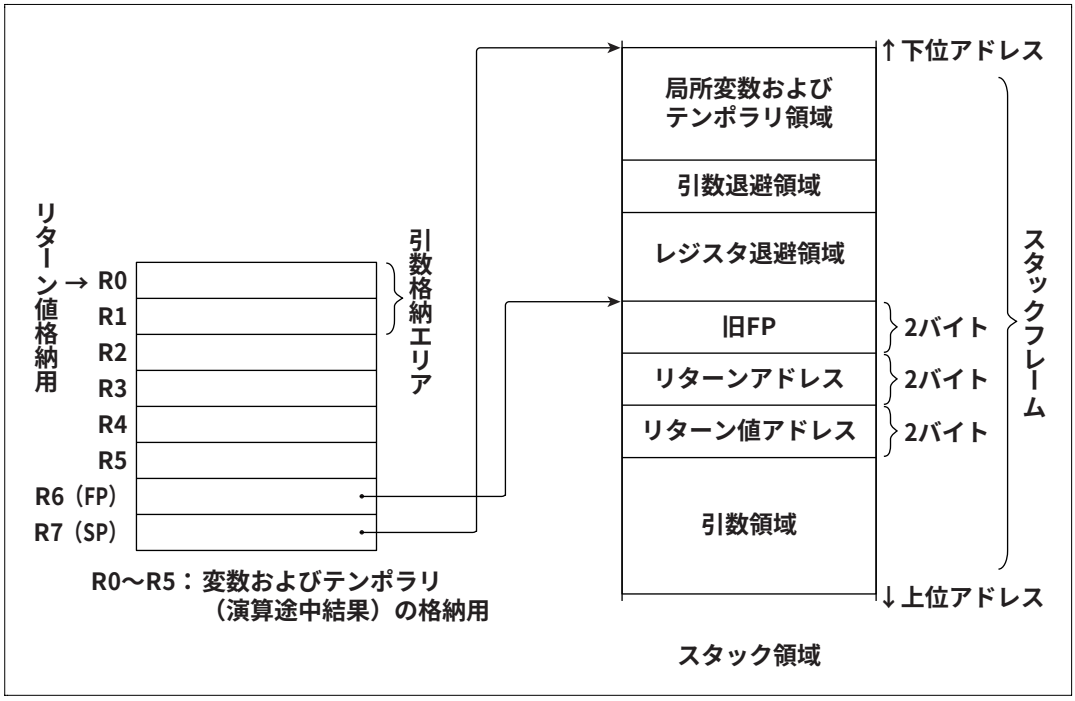


図 C.5　**非最適化時のレジスタとスタック領域の使用法**（H8/300 **用**）

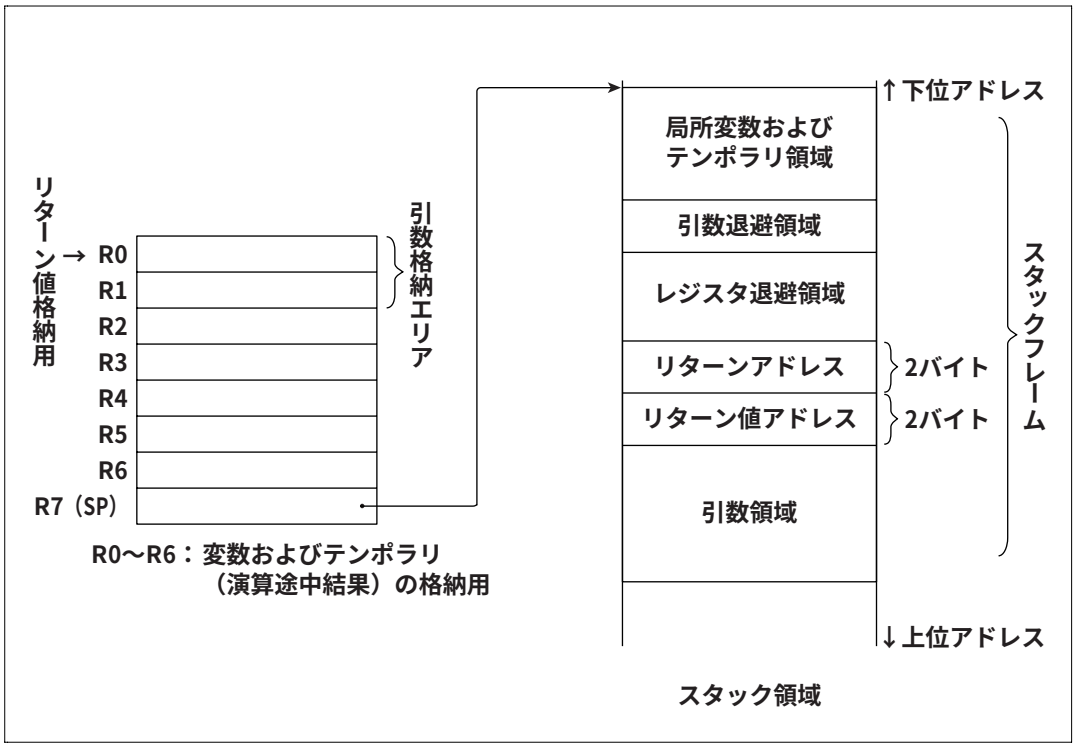


図 C.6　**最適化時のレジスタとスタック領域の使用法**（H8/300 **用**）

D.　終了処理関数の作成例

D.1　終了処理の登録と実行（onexit）ルーチンの作成例

終了処理の登録を行なうライブラリ onexit 関数の作成法を示します。

onexit 関数では、引数として渡された関数のアドレスを、終了処理のテーブルに登録します。登録された関数の個数が限界値（ここでは、登録できる個数を 32 個とします）をこえた場合、あるいは同じ関数が二度以上登録された場合はリターン値として NULL を返します。そうでなければ NULL 以外の値（この場合は、登録した関数のアドレス）を返します。

以下にプログラム例を示します。

■例

```
#include <stdlib.h>
typedef void *onexit_t ;

int _onexit_count=0 ;
onexit_t (*_onexit_buf[32])(void) ;

extern  onexit_t onexit(onexit_t (*)(void)) ;

onexit_t onexit(f)
onexit_t (*f)(void) ;
{
     int i;

     for(i=0; i<_onexit_count ; i++)     /* 既に登録されていないか
                                                    チェック*/
          if(_onexit_buf[i]==f)
               return NULL ;
     if (_onexit_count==32)              /* 登録数の限界値チェック*/
          return NULL ;
     else {
          _onexit_buf[_onexit_count++]=f;/* 関数のアドレスを登録 */
          return f;
     }
}
```

D.2　プログラムの終了（exit）ルーチンの作成例

プログラムの終了処理を行なうライブラリ exit 関数の作成法を示します。プログラムの終了処理は、ユーザシステムによって異なりますので、以下のプログラム例を参考に、ユーザシステムの仕様に従った終了処理を作成してください。

exit 関数は、引数として渡されたプログラムの終了コードに従って C プログラムの終了処理を行ない、プログラム起動時の環境に戻ります。ここでは、終了コードを外部変数に設定して、main 関数を呼び出す直前に setjmp 関数で退避した環境に戻ることによって実現します。

以下にプログラム例を示します。

■例

```
#include <setjmp.h>
#include <stddef.h>

typedef void * onexit_t ;
extern int _onexit_count ;
extern onexit_t (*_onexit_buf[32])(void) ;

extern jmp_buf _init_env ;
extern int _exit_code ;

extern void _CLOSEALL(void);
extern void exit(int);

void exit(code)
int code ;
{
      int i;

      _exit_code=code ;                  /* _exit_code にリターンコードを設定*/

      for(i=_onexit_count-1; i>=0; i--) /* onexit 関数で登録した関数を順次実行*/
            (*_onexit_buf[i])();

      _CLOSEALL();                       /* オープンした関数を全てクローズ*/

      longjmp(_init_env, 1) ;            /* setjmp で退避した環境にリターン*/
}
```

■注意

上記関数で、プログラム実行前の環境に戻るためには、次の関数「callmain」を作成し、初期化ルーチン「init」から関数「main」を呼び出す代わりに関数「callmain」を呼び出してください。

```
#include <setjmp.h>

jmp_buf _init_env;
int     _exit_code;

void callmain()
{

    /*  setjmp を用いて現在の環境を退避し、main 関数を呼び出します。     */
    /*  exit 関数からのリターン時には処理を終了します。                  */

    if(!setjmp(_init_env))
          _exit_code=main();
}
```

D.3 異常終了（abort）ルーチンの作成例

異常終了の場合は、ご使用になっているユーザシステムの仕様に従ってプログラムを異常終了させる処理を行なってください。

以下、標準出力装置にメッセージを出力したあと、ファイルをクローズしてから無限ループしてリセットを待つプログラム例を示します。

■例

```
#include <stdio.h>
extern void abort(void);
extern void _CLOSEALL(void);
void abort()
{
       printf(“program is abort !!¥n”);        /*メッセージ出力    */
       _CLOSEALL();                            /*ファイルのクローズ */
       while(1)   ;                            /*無限ループ        */
}
```

E.　低水準インタフェースルーチンの作成例

E.1　低水準インタフェースルーチン作成例

```
/*************************************************************************/
/*                                lowsrc.c:                              */
/*－－－
```

```
/*****************************************************************************/
/* 静的変数の定義：                                                          */
/* 低水準インタフェースルーチンで使用する静的変数の定義                      */
/*****************************************************************************/

char flmod[FLMAX];                      /* オープンしたファイルのモード設定場所 */

static  union  {
              short  dummy ;            /* 2バイト境界にするためのダミー       */
              char heap[HEAPSIZE];      /* sbrkで管理する領域の宣言            */
             } heap_area ;

static  char  *brk=(char *)&heap_area;  /* sbrkで割り付けた領域の最終アドレス  */

/*****************************************************************************/
/*  open:ファイルのオープン                                                  */
/*       リターン値：ファイル番号（成功）                                    */
/*                   -1          （失敗）                                    */
/*****************************************************************************/
open(char *name,                                /* ファイル名                */
    int  mode,                                  /* ファイルのモード          */
    int  flg)                                   /* 未使用                    */
{
      /* ファイル名に従ってモードをチェックし、ファイル番号を返す              */

      if(strcmp(name,"stdin")==0){              /* 標準入力ファイル          */
            if((mode&O_RDONLY)==0)
                    return -1;
            flmod[STDIN]=mode;
            return STDIN;
      }

      else if(strcmp(name,"stdout")==0){        /* 標準出力ファイル          */
            if((mode&O_WRONLY)==0)
                    return -1;
            flmod[STDOUT]=mode;
            return STDOUT;
      }

      else if(strcmp(name,"stderr")==0){        /* 標準エラー出力ファイル    */
            if((mode&O_WRONLY)==0)
                    return -1;
            flmod[STDERR]=mode;
            return STDERR;
      }

      else
            return -1;                          /* エラー                    */
}
```

```
/*****************************************************************************/
/*  close:ファイルのクローズ                                                 */
/*       リターン値：0          （成功）                                     */
/*                  -1          （失敗）                                     */
/*****************************************************************************/
close(int fileno)                                   /* ファイル番号             */
{
      if(fileno<FLMIN || FLMAX<=fileno)             /* ファイル番号の範囲チェック  */
            return -1;

      flmod[fileno]=0;                              /* ファイルのモードリセット    */
      return 0;
}


/*****************************************************************************/
/*  read:データの読み込み                                                    */
/*        リターン値：実際に読み込んだ文字数  （成功）                       */
/*                  -1                        （失敗）                       */
/*****************************************************************************/
read(int  fileno,                                   /* ファイル番号             */
    char *buf,                                      /* 転送先バッファアドレス       */
    int  count)                                     /* 読み込み文字数             */
{
      int i;

      /* ファイル名に従ってモードをチェックし、一文字づつ入力してバッファに格納     */

      if(flmod[fileno]&O_RDONLY || flmod[fileno]&O_RDWR){
            for(i=count; i>0; i--){
                  *buf=charget();
                  if(*buf==CR)                      /* 改行文字の置き換え          */
                        *buf=LF;
                  buf++;
            }
            return count;
      }
      else
            return -1;
}
```

```
/*****************************************************************************/
/*  write:データの書き出し                                                   */
/*        リターン値：実際に書き出した文字数（成功）                         */
/*                  -1                  （失敗）                             */
/*****************************************************************************/
write(int  fileno,                          /* ファイル番号                  */
     char *buf,                             /* 転送元バッファアドレス        */
     int  count)                            /* 書き出し文字数                */
{
      int  i;
      char c;

      /* ファイル名に従ってモードをチェックし、一文字づつ出力                 */

      if(flmod[fileno]&O_WRONLY || flmod[fileno]&O_RDWR){
            for(i=count; i>0; i--){
                  c=*buf++;
                  charput(c);
            }
            return count;
      }
      else
            return -1;
}
/*****************************************************************************/
/*  lseek:ファイルの読み込み／書き出し位置の設定                             */
/*        リターン値：読み込み／書き出し位置のファイル先頭からのオフセット（成功）*/
/*                  -1          （失敗）                                     */
/*     (コンソール入出力では、lseek はサポートしていません）                 */
/*****************************************************************************/
long lseek(int  fileno,                     /* ファイル番号                  */
          long offset,                      /* 読み込み／書き出し位置        */
          int  base)                        /* オフセットの起点              */
{
      return -1L;
}

/*****************************************************************************/
/*  sbrk:データの書き出し                                                    */
/*       リターン値：割り付けた領域の先頭アドレス（成功）                    */
/*                 -1                     （失敗）                           */
/*****************************************************************************/
char  *sbrk(int size)                       /* 割り付ける領域のサイズ        */
{
     char  *p ;

     if (brk+size>heap_area.heap+HEAPSIZE)  /* 空き領域のチェック            */
          return (char *)-1 ;

     p=brk ;                                /* 領域の割り付け                */
     brk += size ;                          /* 最終アドレスの更新            */
     return p ;
}
```

```
;―――――――――――――――――――――――――――――――――――――――――――――
;                                lowlvl.nor                                 |
;―――――――――――――――――――――――――――――――――――――――――――――
;  H8S, H8/300 シリーズ　シミュレータ・デバッガ　インタフェースルーチン             |
;                        ―一文字入出力を行ないます―                          |
;―――――――――――――――――――――――――――――――――――――――――――――
;  H8S/2600 用、H8S/2000 用、H8/300H 用ノーマルモード(cpu=2600n,cpu=2000n,cpu=300hn)  |
;―――――――――――――――――――――――――――――――――――――――――――――
          .CPU        2600N                     ; または 2000N,300HN
          .EXPORT     _charput
          .EXPORT     _charget

SIM_IO:   .EQU        H'00FE                    ; TRAP_ADDRESS の指定

          .SECTION    P,CODE,ALIGN=2
;―――――――――――――――――――――――――――――――――――――――――――――
;  _charput: 一文字出力                                                      |
;           C プログラムインタフェース：  charput(char)                        |
;―――――――――――――――――――――――――――――――――――――――――――――

_charput:
          MOV.B       R0L,@IO_BUF               ; パラメタをバッファに設定
          MOV.W       #H'0102,R0                ; パラメタ、機能コードの設定
          MOV.W       #LWORD IO_BUF,R1
          MOV.W       R1,@PARM                  ; 入出力バッファアドレスの設定
          MOV.W       #LWORD PARM,R1            ; パラメタブロックアドレスの設定
          JSR         @SIM_IO
          RTS

;―――――――――――――――――――――――――――――――――――――――――――――
;  _charget: 一文字入力                                                      |
;           C プログラムインタフェース：  char charget(void)                   |
;―――――――――――――――――――――――――――――――――――――――――――――

_charget:
          MOV.W       #H'0101,R0                ; パラメタ、機能コードの設定
          MOV.W       #LWORD IO_BUF,R1
          MOV.W       R1,@PARM                  ; 入出力バッファアドレスの設定
          MOV.W       #LWORD PARM,R1            ; パラメタブロックアドレスの設定
          JSR         @SIM_IO
          MOV.B       @IO_BUF,R0L
          RTS

;―――――――――――――――――――――――――――――――――――――――――――――
; 入出力用バッファの定義                                                      |
;―――――――――――――――――――――――――――――――――――――――――――――

          .SECTION    B,DATA,ALIGN=2

PARM:     .RES.W      1                         ; パラメタブロック領域
IO_BUF:   .RES.B      1                         ; 入出力バッファ領域

          .END
```

```
;―――――――――――――――――――――――――――――――――――――――――――――――
;                                lowlvl.adv                                    |
;―――――――――――――――――――――――――――――――――――――――――――――――
;  H8S,H8/300 シリーズ  シミュレータ・デバッガ  インタフェースルーチン          |
;                        ―一文字入出力を行ないます―                           |
;―――――――――――――――――――――――――――――――――――――――――――――――
; H8S/2600 用、H8S/2000 用、H8/300H 用アドバンストモード(cpu=2600a,cpu=2000a,cpu=300ha)|
;―――――――――――――――――――――――――――――――――――――――――――――――
          .CPU        2600A                     ; または 2000A,300HA
          .EXPORT     _charput
          .EXPORT     _charget

SIM_IO:   .EQU        H'01FE                    ; TRAP_ADDRESS の指定

          .SECTION    P,CODE,ALIGN=2

;―――――――――――――――――――――――――――――――――――――――――――――――
;  _charput: 一文字出力                                                        |
;           C プログラムインタフェース：  charput(char)                        |
;―――――――――――――――――――――――――――――――――――――――――――――――

_charput:
          MOV.B       R0L,@IO_BUF               ; パラメタをバッファに設定
          MOV.W       #H'0112,R0                ; パラメタ、機能コードの設定
          MOV.L       #IO_BUF,ER1
          MOV.L       ER1,@PARM                 ; 入出力バッファアドレスの設定
          MOV.L       #PARM,ER1                 ; パラメタブロックアドレスの設定
          JSR         @SIM_IO
          RTS

;―――――――――――――――――――――――――――――――――――――――――――――――
;  _charget: 一文字入力                                                        |
;           C プログラムインタフェース：  char charget(void)                   |
;―――――――――――――――――――――――――――――――――――――――――――――――

_charget:
          MOV.W       #H'0111,R0                ; パラメタ、機能コードの設定
          MOV.L       #IO_BUF,ER1
          MOV.L       ER1,@PARM                 ; 入出力バッファアドレスの設定
          MOV.L       #PARM,ER1                 ; パラメタブロックアドレスの設定
          JSR         @SIM_IO
          MOV.B       @IO_BUF,R0L
          RTS

;―――――――――――――――――――――――――――――――――――――――――――――――
; 入出力用バッファの定義                                                       |
;―――――――――――――――――――――――――――――――――――――――――――――――

          .SECTION    B,DATA,ALIGN=2

PARM:     .RES.L      1                         ; パラメタブロック領域
IO_BUF:   .RES.B      1                         ; 入出力バッファ領域

          .END
```

```
;―――――――――――――――――――――――――――――――――――――――――――――――
;                                lowlvl.reg                                    |
;―――――――――――――――――――――――――――――――――――――――――――――――
;  H8S, H8/300 シリーズ  シミュレータ・デバッガ  インタフェースルーチン          |
;                      ―一文字入出力を行ないます―                             |
;―――――――――――――――――――――――――――――――――――――――――――――――
;                    H8/300 用(cpu=300)                                        |
;―――――――――――――――――――――――――――――――――――――――――――――――
          .CPU        300
          .EXPORT     _charput
          .EXPORT     _charget

SIM_IO:   .EQU        H'00FE                    ; TRAP_ADDRESS の指定

          .SECTION    P,CODE,ALIGN=2

;―――――――――――――――――――――――――――――――――――――――――――――――
;  _charput: 一文字出力                                                        |
;           C プログラムインタフェース：  charput(char)                        |
;―――――――――――――――――――――――――――――――――――――――――――――――

_charput:
          MOV.B       R0L,@IO_BUF               ; パラメタをバッファに設定
          MOV.W       #H'0102,R0                ; パラメタ、機能コードの設定
          MOV.W       #IO_BUF,R1
          MOV.W       R1,@PARM                  ; 入出力バッファアドレスの設定
          MOV.W       #PARM,R1                  ; パラメタブロックアドレスの設定
          JSR         @SIM_IO
          RTS

;―――――――――――――――――――――――――――――――――――――――――――――――
;  _charget: 一文字入力                                                        |
;           C プログラムインタフェース：  char charget(void)                   |
;―――――――――――――――――――――――――――――――――――――――――――――――

_charget:
          MOV.W       #H'0101,R0                ; パラメタ、機能コードの設定
          MOV.W       #IO_BUF,R1
          MOV.W       R1,@PARM                  ; 入出力バッファアドレスの設定
          MOV.W       #PARM,R1                  ; パラメタブロックアドレスの設定
          JSR         @SIM_IO
          MOV.B       @IO_BUF,R0L
          RTS

;―――――――――――――――――――――――――――――――――――――――――――――――
; 入出力用バッファの定義                                                       |
;―――――――――――――――――――――――――――――――――――――――――――――――

          .SECTION    B,DATA,ALIGN=2

PARM:     .RES.W      1                         ; パラメタブロック領域
IO_BUF:   .RES.B      1                         ; 入出力バッファ領域

          .END
```

F. 短絶対アドレスのアクセス範囲

F.1 短絶対アドレスのアクセス範囲

各 CPU／動作モードにおける 8 ビット絶対アドレスおよび 16 ビット絶対アドレスのアクセス範囲を**表** F.1 に示します。

表 F.1 **短絶対アドレスのアクセス範囲**

CPU／**動作モード**	8 **ビット絶対アドレス**（@aa:8）**のアクセス範囲**	16 **ビット絶対アドレス**（@aa:16）**のアクセス範囲**
2600a[:32] 2000a[:32]	0xFFFFFF00〜0xFFFFFFFF	0x0〜0x7FFF、 0xFFFF8000〜0xFFFFFFFF
2600a:28 2000a:28	0xFFFFF00〜0xFFFFFFF	0x0〜0x7FFF、 0xFFF8000〜0xFFFFFFF
2600a:24 2000a:24 300ha[:24]	0xFFFF00〜0xFFFFFF	0x0〜0x7FFF、 0xFF8000〜0xFFFFFF
2600a:20 2000a:20 300ha:20	0xFFF00〜0xFFFFF	0x0〜0x7FFF、 0xF8000〜0xFFFFF
2600n 2000n 300hn 300	0xFF00〜0xFFFF	—

G.　CPU 情報ファイルの作成

モジュール間最適化ツールは、H8S,H8/300 シリーズの CPU ごとのメモリマップに合わせて最適化および再割り付けを行うため、CPU 情報ファイルを使用します。CPU 情報ファイルは、CIA（CPU Information Analyzer）を用いて作成します。本章では、CPU 情報ファイルの作成方法について説明します。

G.1　CIA の機能

CIA は次の３つの機能を持ちます。

(a) CPU 情報ファイルの作成

使用するデバイスのメモリマップ情報をファイルに作成します。

(b) CPU 情報ファイルの内容表示

作成済みの CPU 情報ファイルの内容を確認することができます。

(c) CPU 情報の編集（削除／追加）

作成済みの CPU 情報ファイルの内容を削除・追加機能を用いて変更することができます。

G.2　CIA の起動方法

CIA を起動するためのコマンドフォーマットは次のとおりです。

```
cia38△<CPU 情報ファイル名>[[△]-work=<テンポラリファイル用ドライブ名およびディレクトリ名>]
```

<CPU 情報ファイル名>として新規または既存の CPU 情報ファイル名を指定します。既存の CPU 情報ファイルを指定した場合は、出力用の CPU 情報ファイルを指定するように要求します。<CPU 情報ファイル名>のファイル形式を省略した場合は、".cpu"を仮定します。

<テンポラリファイル用ドライブ名およびディレクトリ名>は、CIA のテンポラリファイルを作成するドライブ名およびディレクトリ名を指定します。オプション省略時は、CIA を起動したディレクトリにテンポラリファイルを作成します。

G.3 CIA の使用手順と選択メニュー

CIA の使用手順を図 G.1 に示します。

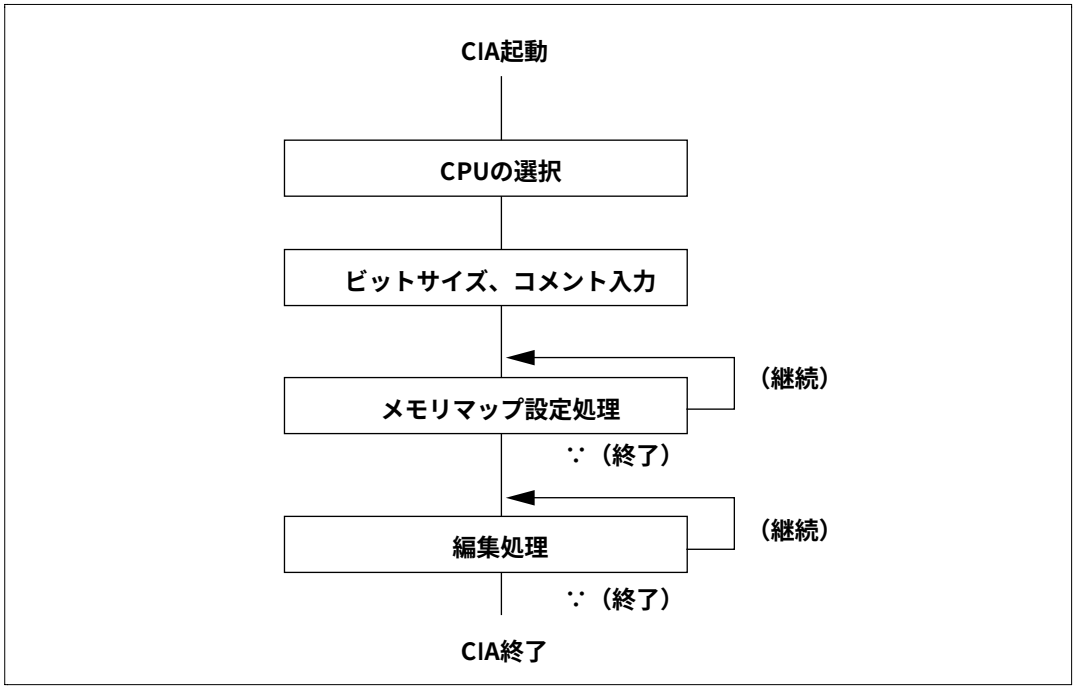


図 G.1 CIA の使用手順

(1) CPU の選択

CPU 情報のメニューとして次の種類があります。

1:H8/300H-ADVANCED　2:H8/300H-NORMAL　3:H8/300　4:H8/300L
5:H8S/2600-ADVANCED　6:H8S/2600-NORMAL
7:H8S/2000-ADVANCED　8:H8S/2000-NORMAL

(2) ビットサイズ、コメント入力

ビットサイズは、メモリマップの設定可能範囲なビット幅を 10 進数で指定します。例えば、16 と指定すると H'0～H'FFFF 番地が設定可能になり、24 と指定すると H'0～H'FFFFFF 番地が設定可能となります。 H8S シリーズでは、さらにプログラム領域とデータ領域の範囲のビット幅を指定することができます。プログラム領域にデータ領域より小さいビット幅を指定した場合、メモリマップの設定可能範囲はデータ領域に指定したビット幅になります。プログラム領域にはデータ領域より大きいビット幅を指定できません。

コメントは、CPU 情報の識別用として、文字列を指定することができます。コメントは、127 文字まで設定できます。

ビットサイズ、コメント入力は、新規ファイルの場合のみです。既存ファイルの場合は、(4)の編集処理から始まります。

（3）メモリマップ設定

CPU 情報の入力のメニューとして次の種類があり、"."(END)が指定されるまでメモリマップの設定処理を繰り返します。

```
0:ROM          1:EXTERNAL     2:RAM        3:I/O        4:EEPROM     .:END
```

0～４はメモリの種別を指定するためのもので、これらを選択した場合には、メモリの開始アドレス、終了アドレス、ステート数、およびデータバス幅を設定します。

アドレスは 16 進数で指定します。指定可能なアドレス範囲は(2)で指定したビットサイズに依存します。

ステート数、データバス幅は 10 進数で指定します。ステート数の指定可能範囲は 1～65535 です。また、データバス幅の指定可能範囲は 8～65528 の８の倍数です。

"." (END)を選択した場合は、入力メニュー処理を終了します。

（4）編集処理

CPU 情報の編集用のメニューとして、次の種類があります。

```
1:ADD        2:DELETE    3:COMMENT     4:CIA ABORT      .:CIA END
```

（a）"1"(ADD)を選択した場合は、(3)のメモリマップ設定処理を行います。

（b）"2" (DELETE)を指定した場合は、消去したいアドレスの範囲を番号で入力します。

（c）"3" (COMMENT)を選択した場合は、新規のコメントを入力します。

（d）"4" (CIA ABORT)を選択した場合は、CPU 情報ファイルに出力せずに、CIA の処理を終了します。

（e）"." (CIA END)を選択した場合は、メモリマップ情報を CPU 情報ファイルに出力し、正常に CIA の処理を終了します。

G.4 CIA の使用例

CIA の使用例を次に示します。下線部はユーザの入力部分です。

(1) H8/310 用 CPU 情報ファイルの作成例

```
>cia38 c310.cpu(RET)          *1

*** NEW FILE ***
   *** CPU MENU ***
   1:H8/300H-ADVANCED   2:H8/300H-NORMAL   3:H8/300   4:H8/300L
   5:H8/2600-ADVANCED   6:H8/2600-NORMAL
   7:H8/2000-ADVANCED   8:H8/2000-NORMAL
   ? 3(RET) *2
BIT SIZE 16 ? : 16(RET) *3
COMMENT?     : '97.03.09 H8/310 SAMPLE(RET) *4
               *** MAP MENU ***
               0:ROM   1:EXTERNAL   2:RAM   3:I/O   4:EEPROM   .:END
               ? 0(RET) *5
                  *  ROM AREA START ADDRESS ?   000000(RET)      *6
                             END   ADDRESS ?   0027FF(RET)      *7
                     STATE COUNT           ?   2(RET)           *8
                     DATA BUS SIZE         ?   16(RET)          *9
                  *  ROM AREA START ADDRESS ?   .(RET)           *10
               *** MAP MENU ***
               0:ROM   1:EXTERNAL   2:RAM   3:I/O   4:EEPROM   .:END
               ? 4(RET)
                  *  EEPROM   START ADDRESS ?   006000(RET)
                             END   ADDRESS ?   007FFF(RET)
                     STATE COUNT           ?   2(RET)
                     DATA BUS SIZE         ?   16(RET)
                  *  EEPROM   START ADDRESS ?   .(RET)
               *** MAP MENU ***
               0:ROM   1:EXTERNAL   2:RAM   3:I/O   4:EEPROM   .:END
               ? 2(RET)
                  *  RAM AREA START ADDRESS ?   00FEC0(RET)
                             END   ADDRESS ?   00FFBF(RET)
                     STATE COUNT           ?   2(RET)
                     DATA BUS SIZE         ?   16(RET)
                  *  RAM AREA START ADDRESS ?   .(RET)
               *** MAP MENU ***
               0:ROM   1:EXTERNAL   2:RAM   3:I/O   4:EEPROM   .:END
               ? 3(RET)
                  *  I/O AREA START ADDRESS ?   00FFF8(RET)
                             END   ADDRESS ?   00FFFF(RET)
                     STATE COUNT           ?   3(RET)
                     DATA BUS SIZE         ?   8(RET)
                  *  I/O AREA START ADDRESS ?   .(RET)
               *** MAP MENU ***
               0:ROM   1:EXTERNAL   2:RAM   3:I/O   4:EEPROM   .:END
               ? .(RET) *11
                     ***** CPU INFORMATION *****
                     CPU : H8/300
                     '91.03.09 H8/310 SAMPLE
                     PROGRAM AREA  BIT SIZE : 16
                     DATA    AREA  BIT SIZE  : 16
                        No    Device     Start     End     State   Bus
                         1 :  ROM AREA : 000000 - 0027FF      2      16
                         2 :  EEPROM   : 006000 - 007FFF      2      16
                         3 :  RAM AREA : 00FEC0 - 00FFBF      2      16
                         4 :  I/O AREA : 00FFF8 - 00FFFF      3       8
      ** EDIT MENU **
      1:ADD    2:DELETE   3:COMMENT  4:CIA ABORT  .:CIA END
      ? .(RET)
*** CIA COMPLETED ***
```

【注】　*1 CIA の起動時に新規作成ファイルを指定します。

*2 CPU の種類を指定します。

*3 ビットサイズを 10 進数で指定します。省略した場合は、表示したデフォルトを設定します。

*4 コメントを入力します。省略した場合は、空白を表示します。128 文字以上入力した場合は、ウォーニングメッセージを出力し、128 文字目以降を無視します。

*5 入力メニューに対し、メモリ種別を番号で入力します。

*6 当該メモリの先頭アドレスを 16 進数で入力します。

*7 当該メモリの終了アドレスを 16 進数で入力します。

*8 当該メモリのステート数を 10 進数で入力します。

*9 当該メモリのデータバス幅を 10 進数で入力します。

*10当該メモリ種別入力の入力を終了したい場合は"."を指定します。

*11入力メニューを終了すると自動的に編集メニューを表示します。

G.5　制限事項一覧

表 G.1 に CIA の制限事項一覧を示します。CIA ではこれらの制限値を超える処理はできません。

表 G-1　CIA の制限事項

No	項目	制限値	備考
1	ビットサイズ	10 進数による指定 指定範囲は 16～32	－
2	アドレス指定	16 進数による指定 指定範囲はビットサイズによる	ビットサイズ 16 の場合 H'0～H'FFFF
3	ステート数	10 進数による指定 指定範囲は 1～65535	ウェートステートを挿入する場合は、ウェートステート数を含めた値を設定してください。
4	データバス幅	10 進数による指定 指定範囲は 8～65528 の８の倍数	－
5	コメントの長さ	127 文字まで	－
6	MAP 情報数	最大 65535 個	但し、CIA が稼動するシステムのメモリ容量によって制限されます。無効領域も情報数に含みます。

G.6 CIA のエラーメッセージ

エラー番号	エラーメッセージ
	エラー内容

表 G.2 CIA のエラーメッセージ

7001	CAN NOT GET MEMORY SPACE
	CIA で使うメモリを確保できません。
7002	CAN NOT OPEN INPUT CPU INFORMATION FILE
	指定された既存の CPU 情報ファイルがオープンできません。
7003	CAN NOT OPEN OUTPUT CPU INFORMATION FILE
	指定された出力用の CPU 情報ファイルがオープンできません。
7004	CAN NOT READ
	ファイルの読み出しができません。
7005	CAN NOT WRITE
	ファイルの書き込みができません。
7006	CAN NOT CLOSE
	出力する CPU 情報ファイルがクローズできません。
7007	INVALID CPU INFORMATION
	CPU 情報ファイルの内容に誤りがあります。
7008	SYNTAX ERROR
	コマンドラインの指定内容に誤りがあります。
8001	COMMENT LINE TOO LONG
	指定したコメントの文字数が 127 文字を超えています。
8002	ADDRESS RE-USE
	アドレス範囲が重複しています。
8003	ADDRESS SIZE OVERFLOW
	ビットサイズを超えたアドレスを指定しています。
8004	INVALID VALUE
	許容範囲外の数値を指定しています。
8005	INVALID CARACTER
	使用できない文字を指定しています。
8006	INVALID END ADDRESS
	終了アドレスが先頭アドレスより小さい値になっています。

H.　旧バージョンとの相違点

新バージョン（H8S, H8/300 シリーズ　C コンパイラ Ver.2.0）と旧バージョン（H8S, H8/300 シリーズ C コンパイラ Ver.1.0）の相違点を示します。

H.1　機能追加・改善項目

（1）モジュール間最適化

C コンパイラおよびアセンブラ出力オブジェクトプログラムを、リンク時にモジュール間にまたがって最適化するモジュール間最適化ツールが追加になりました。従来Ｃコンパイラだけでは最適化できなかった、メモリ配置や関数の呼び出し関係に依存した最適化を実現します。

（2）外部変数のレジスタ割り付け

#pragma global_register (<変数名>=<レジスタ名>)により、大域変数を汎用レジスタに割り付けます。アクセス頻度の高い変数を割り付けることにより、トータルパフォーマンスが向上します。

（3）インラインアセンブラ機能

#pragma inline_asm (<関数名>)により、Ｃソースファイル内でアセンブリ言語で記述した関数をインライン展開します。

（4）高速ライブラリのサポート

標準ライブラリとして、コードサイズ優先とスピード優先の２種類のライブラリを提供します。

（5）標準ライブラリ機能の拡張

入出力(stdio.h)とメモリ割り付け(stdlib.h)のＣライブラリ関数に対して、拡張機能をサポートしました。

・浮動小数点変換をサポートしない簡易入出力関数を提供します。

浮動小数点変換を使用しないファイル入出力を行う場合、ROM サイズを大幅に低減できます。

・入出力ファイル数の可変化

従来 20 個固定だった入出力ファイル数をユーザ指定できるようになりました。

・ヒープ領域確保サイズの可変化

sbrk ルーチンで一回に確保するヒープ領域サイズをユーザ指定できるようになりました。

（6）従来機能の改善

従来からサポートしている以下機能を改善しました。

（a）メモリ間接の関数呼び出し

従来#pragma indirect により、メモリ間接関数呼び出しを指定できる関数はユーザ指定関数のみでしたが、#include <indirect.h>を宣言することにより、実行時ルーチンの呼び出しコードもメモリ間接呼び出しにすることができるようになりました。

（b）インライン展開関数

インライン展開後の関数定義を削除します。

（c）割り込み関数

割り込み関数宣言のあるファイル内でも、ベクタテーブルの設定ができるようになりました。

（d）漢字コード変換

SJIS→EUC、EUC→SJIS の変換が可能になりました。

（e）limits オプションの廃止

コンパイラが、ソースプログラムに合わせて自動的に制限値を拡張できるようになりました。

H.2.　言語仕様上の変更

H.2.1　#pragma 指定

（1）#pragma abs8、#pragma abs16、#pragma indirect

V1.0 台では、#pragma abs8,　#pragma abs16、#pragma indirect 宣言が変数の宣言後にある場合、エラーを出力していましたが、V2.0 以降ではインフォメーションレベルに変更になりました。

■例

```
char a;
#pragma abs8 a
void main(void)
 {
        :
}
```

→ **エラー**2341 **を出力**

V1.0

```
char a;
#pragma abs8 a
void main(void)
{
     :
}
```

→ **インフォメーション** 301 **を出力**

V2.0

（2）サポートしていない #pragma キーワード

V1.0 台では、本コンパイラでサポートしていない#pragma キーワードが指定された場合、エラーを出力していましたが、V2.0 以降ではウォーニングレベルに変更になりました。

（3）#pragma 指定におけるマクロ展開

V2.0 以降では、#pragma 指定のパラメータもマクロ展開の対象になります。

■例

```
#define SUB sub
#pragma inline SUB
void SUB(void)
{
        puts(“****OK!****”);
}
void main(void)
{
      SUB();
}
```

→**インライン展開しない**

V1.0

```
#define SUB sub
#pragma inline SUB
void SUB(void)
{
        puts(“****OK!****”);
}
void main(void)
{
      SUB();
}
```

→**インライン展開する**

V2.0

H.2.2 マクロ展開

(1) Cキーワードのマクロ展開

V1.0 台では、C言語のキーワードをマクロ名として定義できませんでしたが、V2.0 以降ではC言語のキーワードをマクロ名として定義できるようになりました。ただし、予め定義されたマクロ名、キーワード defined および前処理指令行の#に続く識別子は、マクロ名として定義されていてもマクロ名として認識しません。

予め定義されたマクロ名：_LINE_、_FILE_、_DATE_、_TIME_、_STDC_

■例

```
#define define if
#define A     → define は#に続く識別子なので、#if には置換しない
       :
 define(a)    → if(a)に置換
       :
```

(2) マクロ再置換抑止

V2.0 以降では、ANSI 仕様に合わせてマクロの再帰的な置換を抑止しました。

■例

```
#define f(a) a+z                    #define f(a) a+z
#define z    z(z)                   #define z    z(z)
             :                                   :
f(z);        →z(z)(z(z))+z(z)に展開  f(z);       →z(z)+z(z)に展開
             : してエラー出力                    : して正常終了
```

V1.0　　V2.0

H.2.3　その他

（1）ソース行継続文字

V1.0 台では、ソース行継続文字を三文字表記(??/)した場合、エラーとしていましたが、V2.0 以降では継続行として認識できるようになりました。

■例

・V1.0

```
int x=??/→エラー
1;
```

・V2.0

```
int x=??/→正常
1;
```

（2） #if 文のマクロ途中の改行

V1.0 台では、#if 文でマクロを用いた場合、マクロの途中に改行があってもエラーになりませんでしたが、V2.0 以降では ANSI 仕様に合わせてエラーになります。

■例

・V1.0

```
#define f(a,b) (a+b)
int v;
void main(void){
#if f(100,          →正常
         200)
   v=1;
#endif
}
```

・V2.0

```
#define f(a,b) (a+b)
int v;
void main(void){
#if f(100,          →エラー
         200)
   v=1;
#endif
}
```

I.　ASCII コード一覧表

I.1　ASCII コード一覧表

表 I.1　ASCII コード一覧表

下位 4 ビット	上位 4 ビット							
	0	1	2	3	4	5	6	7
0	NUL	LE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	“	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	‘	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	＊	：	J	Z	j	z
B	VT	ESC	＋	;	K	[	k	{
C	FF	FS	,	＜	L	＼	l	|
D	CR	GS	－	＝	M	]	m	}
E	SO	RS	・	＞	N	^	n	～
F	SI	US	／	?	O		o	DEL

索引

日本語索引

あ行

アセンブラ埋め込み....64
アセンブラ埋め込みインライン展開....70
アセンブリプログラムとの結合....52
　外部名の相互参照方法....52
　関数呼び出しのインタフェース....53
　スタックポインタに関する規則....54
　引数とリターン値の設定、参照に関する規則....56
　レジスタに関する規則....55
　スタックフレームの割り付け、解放に関する規則....54
インクルードファイル
　CH38（環境変数）....30
　include（コンパイラオプション）....19
　インクルードファイル基点ディレクトリ....20
　インクルードファイルの検索方法....203
　標準インクルードファイル....4
インターナルレベル（エラーメッセージレベル）....139
インフォメーションレベル（エラーメッセージレベル）....139, 183
インライン展開
　#pragma inline....68
　#pragma inline_asm....70
　inline（コンパイラサブオプション）....12
　nestinline（コンパイラオプション）....13
ウォーニングレベル（エラーメッセージレベル）....139,183
エクステンドレジスタ....85
エラー情報（コンパイルリスト）....34

エラーメッセージ
CIA のエラーメッセージ....248
C コンパイラのエラーメッセージ....139
C ライブラリ関数のエラーメッセージ....160
エラーレベル（エラーメッセージレベル）....139,183
モジュール間最適化のエラーメッセージ....183
オブジェクト情報（コンパイルリスト）....37
オブジェクトプログラムの構造....45
オプション（C コンパイラ）....8
オプション指定方法....5
オプション（モジュール間最適化）....171
オプション指定方法....171

か行

外部名　....52
拡張機能....62
#pragma 拡張子....62
組み込み関数....81
仮数部　....209
簡易入出力関数....94
環境変数....29
CH38（環境変数）....30
CH38TMP（環境変数）....30
H38CPU（環境変数）....29
path（環境変数）....29
関数のインライン展開....68
関数呼び出しのインタフェース....53
起動方法（コンパイラ）....5
起動方法（モジュール間最適化）....168
旧バージョンとの相違点....249
機能追加・改善項目....249
言語仕様上の変更....251
共通式最適化....27
境界調整数....46, 47, 48, 49
共用体型....49 ,201
グローバル変数のレジスタ割り付け....79
組み込み関数....81

限界値 43
言語仕様 195
翻訳の仕様 195
環境の仕様 195
識別子の仕様 195
文字の仕様 196
整数の仕様 197
整数型とその値の範囲 197
浮動小数点数の仕様 198
浮動小数点数の限界値 198
配列とポインタの仕様 199
レジスタの仕様 200
構造体、共用体、列挙型、ビットフィールドの仕様 201
修飾子の仕様 201
宣言の仕様 202
文の仕様 202
プリプロセッサの仕様 203
C ライブラリ関数の仕様 204
stddef.h の仕様 204
assert.h の仕様 204
ctype.h の仕様 205
math.h の仕様 205
stdio.h の仕様 206
string.h の仕様 207
サポートしていないライブラリ 207
浮動小数点数演算の仕様 214
構造体型 49, 201
コマンドラインの形式（コンパイラ） 5
コマンドラインの形式（モジュール間最適化） 168
コーディング上の注意事項 95
コンディションコード反映演算 91
コンディションコードレジスタの設定・参照 83
コンパイルリスト 32

さ行

サイズの算出法 104, 108
スタック領域サイズの算出法 108

静的領域サイズの算出法.....104
ヒープ領域サイズの算出法.....111
最適化.....12, 165
最適化内容の指定（コンパイラ）.....12
最適化レベル（コンパイラ）.....11
最適化内容の指定（モジュール間最適化）.....173
モジュール間最適化.....14, 165
最適化情報（モジュール間最適化）.....178
サブコマンドファイル.....179
指数部.....209
システム組み込み.....103
メモリ領域の割り付け.....104
C ライブラリ関数の実行環境の設定.....120
実行環境の設定.....113
実行時ルーチン.....105
初期化データ領域.....46
初期設定ルーチンの作成.....116
セクションの初期化.....116
ベクタテーブルの設定.....114
終了処理関数の作成例.....231
abort 関数.....234
exit 関数.....231
onexit 関数.....231
シンボル割り付け情報（コンパイルリスト）.....35
スカラ型の内部表現.....48
スタックフレーム.....54, 226
スタックポインタ.....54
スタック領域.....46, 108
下位アドレス.....54
上位アドレス.....54
スタック領域サイズの算出法.....108
正規化数.....210
静的領域サイズの算出法.....104
静的領域の割り付け.....104
セクション.....45
セクションの初期化.....116
初期化データ領域.....46
定数領域.....46

プログラム領域.....46
未初期化データ領域.....46
セクション切り替え.....76
ソースリスト情報（コンパイルリスト）.....32

た行

短絶対アドレス.....23, 63
短絶対アドレスのアクセス範囲.....242
注意事項.....95
CPU／動作モードの選択に関する注意事項.....97
コーディング上の注意事項.....95
トラブル発生時の対処方法.....98
ビット操作命令に関する注意事項.....98
プログラム開発上の注意事項.....97
低水準インタフェースルーチン.....128
lseek ルーチン.....135, 238
close ルーチン.....132, 237
open ルーチン.....130, 236
read ルーチン.....133, 237
sbrk ルーチン.....136, 238
write ルーチン.....134, 238
低水準インタフェースルーチンの作成例.....235
定数領域.....46
データの内部表現.....47
デバッグ情報.....14
動的領域.....46, 108
スタック領域.....108
動的領域の割り付け方.....112
ヒープ領域.....111
特殊命令.....87
トラブル発生時の対処方法.....98
統計情報（コンパイルリスト）.....39

な行

内部表現.....47
入出力.....103

入出力の考え方 128
標準入出力の初期設定 124
低水準インタフェースルーチン 128

は行

配列型 49
引数 56
型変換の規則 57
スタック上の引数領域 60
引数格納用レジスタ 59
引数の渡し方 56
引数の割り付け領域 58
引数割り付けの具体例 218
非正規化数 210
非数 210
ビットフィールド 50, 201
ヒープ領域 46, 111
標準インクルードファイル 4
標準ライブラリ 31
ファイル拡張子 7
ファイルの入出力 128
close ルーチン 132, 237
open ルーチン 130, 236
read ルーチン 133, 237
sbrk ルーチン 136, 238
write ルーチン 134, 238
lseek ルーチン 135, 238
ファイル名の付け方 7
フェータルレベル（エラーメッセージレベル） 139, 183
複合型 49
符号拡張 50
符号部 214
浮動小数点数の仕様 198, 209
浮動小数点数の限界値 198
浮動小数点数の内部表現 209
符号部 209
仮数部 209

指数部 209
浮動小数点数の表現する値の種類 210
正規化数 210
非正規化数 210
無限大 210
非数 210
浮動小数点数演算の仕様 214
丸め 214
オーバフロー 215
アンダフロー 215
無効演算 214
特殊値の演算に関する注意事項 215
10 進演算と内部表現 215
プログラムの開発手順 4, 167
プログラム開発上の注意事項 97
プログラム領域 46
プログラムの構成例 113, 120
ベクタテーブル 114, 121
ポインタ 48

ま行

マクロ名の定義 18
丸め 214
未初期化データ領域 46
無限大 210
無効演算 215
メモリ間接の関数呼び出し 25, 66
メモリ領域の割り付け 104
ROM、RAM の割り付け 106
サイズの算出法 104,108
初期化データ領域の割り付け 106
動的領域の割り付け 108
静的領域の割り付け 104
メモリ割り付け例 107
モジュール間最適化 14, 165
モジュール間最適化のエラーメッセージ 183
文字列出力領域 21

や行

予約語 128

ら行

ライブラリ
C ライブラリ関数 4, 105, 204
C ライブラリ関数の初期設定 123
C ライブラリ関数の実行環境の設定 120
実行時ルーチン 4, 105
標準ライブラリ 4, 31, 105
低水準インタフェースルーチン 128
ライブラリの初期化 123
ライブラリのエラーメッセージ 160
リスト 32
エラー情報 34
オブジェクト情報 37
シンボル割り付け情報 35
ソースリスト情報 32
統計情報 39
リストファイルの構成 32
リターン値に対する一般的な規則 56
リターン値アドレス 60
リターン値格納用レジスタ 61
リターン値の設定場所 60
レジスタ退避／回復コード制御 78
レジスタの仕様 200
引数格納用レジスタ 59
リターン値格納用レジスタ 61
レジスタとスタック領域の使用法 226
レジスタ保証規則 55
レジスタ変数の拡張 27
グローバル変数のレジスタ割り付け 79
ローテート演算 90

わ行

割り込み関数の作成 71

英数字索引

10 進演算（組み込み関数）.....93
2000a（コンパイラサブオプション）.....10
2000n（コンパイラサブオプション）.....10
2600a（コンパイラサブオプション）.....10
2600n（コンパイラサブオプション）.....10
300（コンパイラサブオプション）.....10
300ha（コンパイラサブオプション）.....10
300hn（コンパイラサブオプション）.....10
300l（コンパイラサブオプション）.....10
300reg（コンパイラサブオプション）.....10

A

abort 関数（終了処理関数）.....234
abs16（コンパイラオプション）.....23
abs8（コンパイラオプション）.....23
absolute_fobid（モジュール間最適化オプション／サブコマンド）.....178
allocation（コンパイラサブオプション）.....15
and_ccr（組み込み関数）.....84
and_exr（組み込み関数）.....86
ASCII コード.....254
asmcode（コンパイラサブオプション）.....18

B

branch（モジュール間最適化オプションパラメタ）.....174
bss（コンパイラサブオプション）.....20
byteenum（コンパイラオプション）.....23

C

C ライブラリ関数 105, 204
C ライブラリ関数の実行環境の設定 120
低水準インタフェースルーチン 128
ファイルのクローズ 127
C ライブラリ関数のエラーメッセージ 160
case（コンパイラオプション） 24
chgincpath（コンパイラオプション） 20
CH38 30
CH38TMP 30
char 型 48, 197
cia38 243
close ルーチン（低水準インタフェースルーチン） 132, 237
cmncode（コンパイラオプション） 27
code（コンパイラオプション） 18
comment（コンパイラオプション） 21
const（コンパイラサブオプション） 20, 21
const 型 97
cpu（コンパイラオプション） 10
CPU／動作モード 10
H8/300 10
H8/300H 用アドバンストモード 10
H8/300H 用ノーマルモード 10
H8S/2000 用アドバンストモード 10
H8S/2000 用ノーマルモード 10
H8S/2600 用アドバンストモード 10
H8S/2600 用ノーマルモード 10
cpuexpand（コンパイラオプション） 26
CPU 情報ファイル 243

D

dadd（組み込み関数） 93
data（コンパイラサブオプション） 20, 21
debug（コンパイラオプション） 14
define（コンパイラオプション） 18

double 型.....48, 198, 213
dsub（組み込み関数）.....94

E

eepmov（組み込み関数）.....89
eepmov（コンパイラオプション）.....25
enum 型 23, 48
errno 123
euc（コンパイラオプション）.....21
euc（コンパイラサブオプション）.....22
exit 関数（終了処理関数）.....231
expansion（コンパイラサブオプション）.....15

F

function_call（モジュール間最適化オプションパラメタ）.....174
function_forbid（モジュール間最適化オプション／サブコマンド）.....177
float 型 48, 95, 198, 211

G

get_ccr（組み込み関数）.....84
get_exr（組み込み関数）.....86
get_imask_ccr（組み込み関数）.....84
get_imask_exr（組み込み関数）.....85
goptimize（コンパイラオプション）.....14

H

H8S/2000 シリーズ..... 3
H8S/2600 シリーズ..... 3
H8/300H シリーズ..... 3
H8/300L シリーズ..... 3
H8/300 シリーズ..... 3
H38CPU（コンパイラ環境変数）..... 11, 29

I

information（モジュール間最適化オプション／サブコマンド）.....178
ifthen（コンパイラサブオプション）.....24
include（コンパイラオプション）.....19
indirect（コンパイラオプション）.....25
indirect.h.....67
inline（コンパイラサブオプション）.....12
int 型.....48, 197

L

length（コンパイラサブオプション）.....15
list（コンパイラオプション）.....16
loop（コンパイラサブオプション）.....12
long double 型.....48, 198, 213
long 型.....48, 197
lseek ルーチン.....135, 238

M

mac（組み込み関数）.....89
machinecode（コンパイラサブオプション）.....18
machine.h.....83
macl（組み込み関数）.....89
memmove 関数（C ライブラリ関数）.....208
message（コンパイラオプション）.....17
movfpe（組み込み関数）.....88
movtpe（組み込み関数）.....88

N

nestinline（コンパイラオプション）.....13
nop（組み込み関数）.....90

O

object（コンパイラオプション）.....17

object（コンパイラサブオプション）.....15
onexit 関数（終了処理関数）.....231
open ルーチン.....130, 236
optimize（コンパイラオプション）.....11
optimize（モジュール間最適化オプション／サブコマンド）.....173
optlnk38（モジュール間最適化コマンド）.....168
or_ccr（組み込み関数）.....84
or_exr（組み込み関数）.....86
outcode（コンパイラオプション）.....22
ovfaddc（組み込み関数）.....91
ovfaddl（組み込み関数）.....91
ovfaddw（組み込み関数）.....91
ovfnegc（組み込み関数）.....93
ovfnegl（組み込み関数）.....93
ovfnegw（組み込み関数）.....93
ovfshalc（組み込み関数）.....92
ovfshall（組み込み関数）.....92
ovfshalw（組み込み関数）.....92
ovfsubc（組み込み関数）.....92
ovfsubl（組み込み関数）.....92
ovfsubw（組み込み関数）.....92

P

path 29
#pragma abs16.....63
#pragma abs16 section 76
#pragma abs8 63
#pragma abs8 section.....76
#pragma asm.....64
#pragma endasm.....64
#pragma global_register.....79
#pragma indirect.....66
#pragma indirect section 76
#pragma inline.....68
#pragma inline_asm.....70
#pragma interrupt.....71
#pragma noregsave 78

#pragma regsave....78
#pragma section....76
program（コンパイラサブオプション）....20

R

read ルーチン（低水準インタフェースルーチン）....133, 237
regexpansion（コンパイラオプション）....27
register（コンパイラサブオプション）....12
register（モジュール間最適化オプションパラメタ）....174
ROM、RAM の割り付け....106
ROM 化....106
rotlc（組み込み関数）....90
rotll（組み込み関数）....90
rotlw（組み込み関数）....90
rotrc（組み込み関数）....91
rotrl（組み込み関数）....91
rotrw（組み込み関数）....91

S

safe（モジュール間最適化オプションパラメタ）....174
same_code（モジュール間最適化オプションパラメタ）....174
samecode_forbid（モジュール間最適化オプション／サブコマンド）....176
samesize（モジュール間最適化オプション／サブコマンド）....175
sbrk ルーチン（低水準インタフェースルーチン）....136, 238
section（コンパイラオプション）....20
set_ ccr（組み込み関数）....84
set_exr（組み込み関数）....86
set_imask_ccr（組み込み関数）....83
set_imask_exr（組み込み関数）....85
shift（コンパイラサブオプション）....12
short 型....48, 197
show（コンパイラオプション）....15
signed....48, 197
sjis（コンパイラオプション）....21
sjis（コンパイラサブオプション）....22

sleep（組み込み関数）.....87
source（コンパイラサブオプション）.....15
speed（コンパイラオプション）.....12
speed（モジュール間最適化オプションパラメタ）.....174
statistics（コンパイラサブオプション）.....15
string（コンパイラオプション）.....21
string_unify（モジュール間最適化オプションパラメタ）.....173
struct（コンパイラサブオプション）.....12
subcommand（コンパイラオプション）.....22
subcommand（モジュール間最適化オプション）.....179
switch（コンパイラサブオプション）.....12
symbol_delete（モジュール間最適化オプションパラメタ）.....173
symbol_forbid（モジュール間最適化オプション／サブコマンド）.....176

T

table（コンパイラサブオプション）.....24
tas（組み込み関数）.....88
trapa（組み込み関数）.....87

U

unsigned 48, 197

V

variable_access（モジュール間最適化オプションパラメタ）.....173
variable_forbid（モジュール間最適化オプション／サブコマンド）.....177
volatile（コンパイラオプション）.....28

W

width（コンパイラサブオプション）.....15
write ルーチン（低水準インタフェースルーチン）.....134, 238

X

xor_ccr（組み込み関数）.....85
xor_exr（組み込み関数）.....87